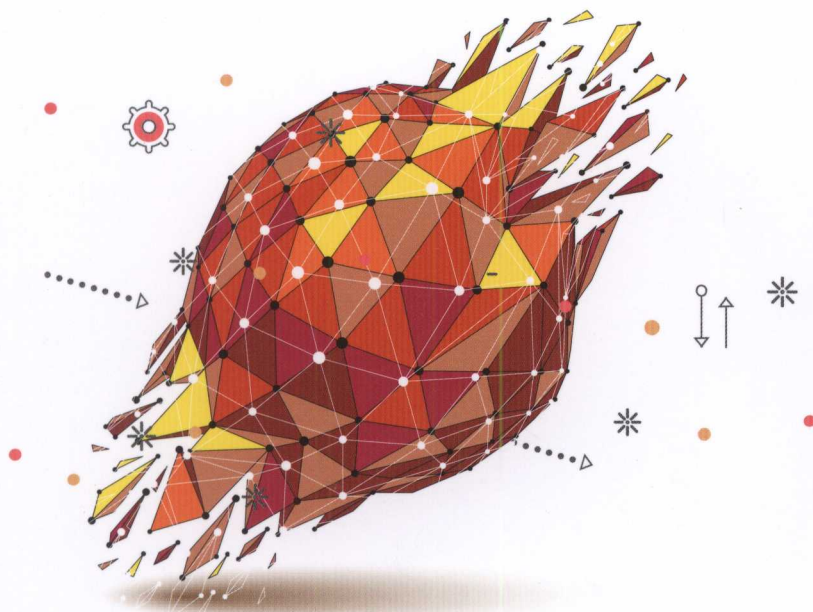


版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



深度学习

核心技术与实践

猿辅导研究团队 著

作者团队

猿辅导研究团队成立于2014年年中，一直从事深度学习在教育领域的应用和研究工作。团队成员均毕业于北京大学、清华大学、上海交大、中科院、香港大学等知名高校，大多数拥有硕士或博士学位。研究方向涵盖了图像识别、语音识别、自然语言理解、数据挖掘、深度学习等领域。团队成功运用深度学习技术，从零开始打造出活跃用户过亿的拍照搜题APP——小猿搜题，开源了分布式机器学习系统ytk-learn和分布式通信系统ytk-mp4j。此外，团队自主研发的一系列成果均成功应用到猿辅导公司的产品中。包括：速算应用中的在线手写识别、古诗词背诵中的语音识别、英语口语智能批改、英文手写拍照识别和英语作文智能批改等技术。

博文视点AI系列

深度学习

核心技术与实践

猿辅导研究团队 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书主要介绍深度学习的核心算法,以及在计算机视觉、语音识别、自然语言处理中的相关应用。本书的作者们都是业界一线的深度学习从业者,所以书中所写内容和业界联系紧密,所涵盖的深度学习相关知识点比较全面。本书主要讲解原理,较少贴代码。

本书适合深度学习从业人士或者相关研究生作为参考资料,也可以作为入门教程来大致了解深度学习的相关前沿技术。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

深度学习核心技术与实践/猿辅导研究团队著.—北京:电子工业出版社,2018.2

(博文视点 AI 系列)

ISBN 978-7-121-32905-0

I. ①深…II. ①猿…III. ①机器学习 IV. ①TP181

中国版本图书馆 CIP 数据核字(2017)第 258324 号

策划编辑:张春雨

责任编辑:葛 娜

印 刷:北京京科印刷有限公司

装 订:北京京科印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:33 字数:718 千字

版 次:2018 年 2 月第 1 版

印 次:2018 年 2 月第 1 次印刷

定 价:119.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819 faq@phei.com.cn。

前言

本书的大部分作者在深度学习流行之前有幸从事机器学习相关工作多年。在我们内部，一直认同一个段子：有多少人工就有多少智能。

- 在深度学习流行之前的传统机器学习年代，我们认为“人工”更多强调的是特征工程之难，需要机器学习从业者不断分析数据，挖掘新的特征。
- 在深度学习流行的这几年，我们认为这句话依然成立，只是“人工”更多地强调人工标注，因为深度学习需要大量的标注数据。当然，也有人反驳说不需要标注，用户的使用历史天然就是标注。实际上，这可以理解为一种众筹标注。
- 在深度学习发展的未来，我们希望这句话不再成立，期待无监督模型取得更长足的进步，使得“人工”智能变为真正的智能。

在追求智能的路上，我们虽然是创业公司，但一直坚持机器学习相关课程的学习和 Paper Reading，陆续学习了传统的机器学习相关算法，也探索了深度学习的相关原理，并不断应用到实践中。

受益于当今学术开放开源的氛围，深度学习的最新算法甚至代码实践大家都能第一时间进行学习。所以在创业公司的早期深度学习实践中，最重要的并不是算法理论方面的创新，而是结合产品需求如何进行深度学习技术的落地。这就要求团队不仅需要对业务非常熟悉，也需要对深度学习相关算法了如指掌，同时还需要有人可以真正用代码将算法落地。很幸运，我们的团队具备这样的能力，所以在深度学习的实践中较少走弯路。随着多年的积累，团队在深度学习方面开始有不少自己的创新，也对理论有了整体的认识。从 2016 年下半年开始，团队部分成员利用周末等业余时间撰写了这本书，算是对团队过去所学深度学习知识的一个总结。本书的撰写都是大家牺牲周末时间完成的，且在撰写过程中，碰到多次项目进度非常紧急的情况，周末时间也被项目占用，但大家还是克服困难，完成了书稿，非常感谢这些作者的配合！此外，猿辅导研究团队的大部分成员参与了审稿相关工作，在此一并表示感谢！

当然，本书撰写较仓促，作者人数也较多，错误和不足在所难免，烦请读者及时反馈，我们将及时纠正。

在这个过程中，有了一点点微不足道的积累。希望通过本书，对过去学过的知识做一些总结归纳，同时分享出来让更多的深度学习爱好者一起受益。

写作分工

朱珊珊编写了第1章的1.2.1节主要部分、1.3节，第2章的绝大部分内容，第13章。

邓澍军编写了前言，第1章的1.1节、1.2.2节至1.2.4节，第2章的2.2.2节、2.2.6节至2.2.8节，第3章，第6章的6.1节、6.2节，第7章，第8章的8.1节，第9章，第10章的10.6节、10.7节，第11、17、18、21章，第25章的25.3节。

陈孟阳编写了第4、10章。

孙萌编写了第5、22章。

冯超编写了第6章的6.3节至6.10节，第8章的8.3节、8.5节至8.7节，第27、28章。

曹月恬编写了第8章的8.2节、8.4节，第24章。

杨晓庆编写了第12、26章。

夏龙编写了第14、15章，第16章的16.1节、16.2节、16.5节、16.6节。

吴凡编写了第16章的16.3节、16.4节。

赵薇编写了第19章。

陈冬晓编写了第20章。

赵玲玲编写了第23章。

王锐坚编写了第25章。

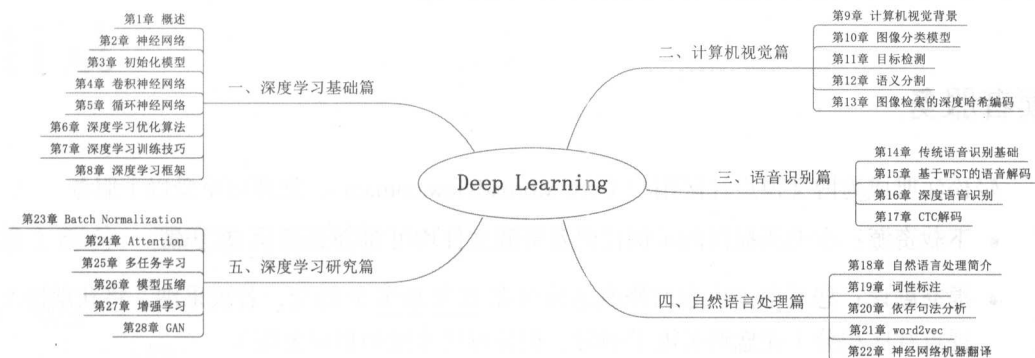
本书特点

本书首先介绍了深度学习的一些基本原理，然后介绍了计算机视觉、语音识别、自然语言处理的相关应用，最后介绍了一些较前沿的研究方向。

本书具有如下特点：

- 计算机视觉、语音识别、自然语言处理这三方面的介绍内容绝大部分是作者团队有过相关实践和研究的方向，和业界联系紧密。
- 所涵盖的深度学习相关知识点比较全面。
- 干货：主要讲解原理，较少贴代码。

本书的篇章脉络如下：



本书读者

本书适合深度学习从业人士或者相关研究生作为参考资料，也可以作为入门教程大致了解深度学习的相关前沿技术。

关于团队

猿辅导研究团队成立于 2014 年，是创业公司中较早从事深度学习的团队。该团队陆续将深度学习应用于如下领域：

- 拍照印刷体 OCR（Optical Character Recognition，光学字符识别）：从 0 开始打造拍照搜题 APP 小猿搜题（目前累计安装量达 1.6 亿次）。
- 拍照手写体 OCR：包括斑马速算产品中的屏幕手写笔迹的在线手写识别、拍照手写图片的离线手写识别、与公务员考试相关的申论手写识别等。
- 语音识别：包括古诗词背诵、高考听说自动判卷、英语口语打分等项目。
- 自然语言处理：主要应用于英语作文自动批改、自动判卷、短文本对话等项目。

关于公司

猿辅导公司是中国领先的移动在线教育机构，拥有中国最多的中学生移动用户，以及国内最大的中学生练习行为数据库，旗下有猿题库、小猿搜题、猿辅导三款移动教育 APP。

2017年6月猿辅导获得由华平投资集团领投、腾讯跟投的1.2亿美元E轮融资，估值超过10亿美元，成为国内K-12在线教育领域首个独角兽公司。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **下载资源：**本书所提供的示例代码及资源文件均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32905>



目录

第 1 部分 深度学习基础篇	1
1 概述	2
1.1 人工智能	3
1.1.1 人工智能的分类	3
1.1.2 人工智能发展史	3
1.2 机器学习	7
1.2.1 机器学习的由来	7
1.2.2 机器学习发展史	9
1.2.3 机器学习方法分类	10
1.2.4 机器学习中的基本概念	11
1.3 神经网络	12
1.3.1 神经网络发展史	13
参考文献	16
2 神经网络	17
2.1 在神经科学中对生物神经元的研究	17
2.1.1 神经元激活机制	17
2.1.2 神经元的特点	18
2.2 神经元模型	19
2.2.1 线性神经元	19
2.2.2 线性阈值神经元	19
2.2.3 Sigmoid 神经元	21
2.2.4 Tanh 神经元	22
2.2.5 ReLU	22
2.2.6 Maxout	24
2.2.7 Softmax	24

2.2.8	小结	25
2.3	感知机	27
2.3.1	感知机的提出	27
2.3.2	感知机的困境	28
2.4	DNN	29
2.4.1	输入层、输出层及隐层	30
2.4.2	目标函数的选取	30
2.4.3	前向传播	32
2.4.4	后向传播	33
2.4.5	参数更新	35
2.4.6	神经网络的训练步骤	36
	参考文献	36
3	初始化模型	38
3.1	受限玻尔兹曼机	38
3.1.1	能量模型	39
3.1.2	带隐藏单元的能量模型	40
3.1.3	受限玻尔兹曼机基本原理	41
3.1.4	二值 RBM	43
3.1.5	对比散度	45
3.2	自动编码器	47
3.2.1	稀疏自动编码器	48
3.2.2	降噪自动编码器	48
3.2.3	栈式自动编码器	49
3.3	深度信念网络	50
	参考文献	52
4	卷积神经网络	53
4.1	卷积算子	53
4.2	卷积的特征	56
4.3	卷积网络典型结构	59
4.3.1	基本网络结构	59
4.3.2	构成卷积神经网络的层	59
4.3.3	网络结构模式	60

4.4	卷积网络的层	61
4.4.1	卷积层	61
4.4.2	池化层	66
	参考文献	67
5	循环神经网络	68
5.1	循环神经网络简介	68
5.2	RNN、LSTM 和 GRU	69
5.3	双向 RNN	75
5.4	RNN 语言模型的简单实现	76
	参考文献	79
6	深度学习优化算法	80
6.1	SGD	80
6.2	Momentum	81
6.3	NAG	82
6.4	Adagrad	84
6.5	RMSProp	85
6.6	Adadelta	86
6.7	Adam	87
6.8	AdaMax	89
6.9	Nadam	89
6.10	关于优化算法的使用	91
	参考文献	91
7	深度学习训练技巧	93
7.1	数据预处理	93
7.2	权重初始化	94
7.3	正则化	95
7.3.1	提前终止	95
7.3.2	数据增强	95
7.3.3	L2/L1 参数正则化	97
7.3.4	集成	99
7.3.5	Dropout	100

参考文献	101
8 深度学习框架	102
8.1 Theano	102
8.1.1 Theano	102
8.1.2 安装	103
8.1.3 计算图	103
8.2 Torch	104
8.2.1 概述	104
8.2.2 安装	105
8.2.3 核心结构	106
8.2.4 小试牛刀	109
8.3 PyTorch	112
8.3.1 概述	112
8.3.2 安装	112
8.3.3 核心结构	113
8.3.4 小试牛刀	113
8.4 Caffe	116
8.4.1 概述	116
8.4.2 安装	117
8.4.3 核心组件	118
8.4.4 小试牛刀	124
8.5 TensorFlow	124
8.5.1 概述	124
8.5.2 安装	124
8.5.3 核心结构	125
8.5.4 小试牛刀	126
8.6 MXNet	130
8.6.1 概述	130
8.6.2 安装	130
8.6.3 核心结构	130
8.6.4 小试牛刀	132
8.7 Keras	134

8.7.1 概述	134
8.7.2 安装	135
8.7.3 模块介绍	135
8.7.4 小试牛刀	135
参考文献	138

第 2 部分 计算机视觉篇 139

9 计算机视觉背景	140
9.1 传统计算机视觉	140
9.2 基于深度学习的计算机视觉	144
9.3 参考文献	145
10 图像分类模型	146
10.1 LeNet-5	146
10.2 AlexNet	148
10.3 VGGNet	153
10.3.1 网络结构	154
10.3.2 配置	156
10.3.3 讨论	156
10.3.4 几组实验	157
10.4 GoogLeNet	158
10.4.1 NIN	160
10.4.2 GoogLeNet 的动机	160
10.4.3 网络结构细节	161
10.4.4 训练方法	163
10.4.5 后续改进版本	164
10.5 ResNet	164
10.5.1 基本思想	164
10.5.2 网络结构	166
10.6 DenseNet	168
10.7 DPN	169
参考文献	169

11 目标检测	172
11.1 相关研究	174
11.1.1 选择性搜索	174
11.1.2 OverFeat	176
11.2 基于区域提名的方法	178
11.2.1 R-CNN	178
11.2.2 SPP-net	180
11.2.3 Fast R-CNN	181
11.2.4 Faster R-CNN	183
11.2.5 R-FCN	184
11.3 端到端的方法	185
11.3.1 YOLO	185
11.3.2 SSD	186
11.4 小结	187
参考文献	189
12 语义分割	191
12.1 全卷积网络	192
12.1.1 FCN	192
12.1.2 DeconvNet	194
12.1.3 SegNet	196
12.1.4 DilatedConvNet	197
12.2 CRF/MRF 的使用	198
12.2.1 DeepLab	198
12.2.2 CRFasRNN	200
12.2.3 DPN	202
12.3 实例分割	204
12.3.1 Mask R-CNN	204
参考文献	205
13 图像检索的深度哈希编码	207
13.1 传统哈希编码方法	207
13.2 CNNH	208
13.3 DSH	209

13.4 小结	211
参考文献	211
第 3 部分 语音识别篇	213
14 传统语音识别基础	214
14.1 语音识别简介	214
14.2 HMM 简介	215
14.2.1 HMM 是特殊的混合模型	217
14.2.2 转移概率矩阵	218
14.2.3 发射概率	219
14.2.4 Baum-Welch 算法	219
14.2.5 后验概率	223
14.2.6 前向-后向算法	223
14.3 HMM 梯度求解	226
14.3.1 梯度算法 1	227
14.3.2 梯度算法 2	229
14.3.3 梯度求解的重要性	233
14.4 孤立词识别	233
14.4.1 特征提取	233
14.4.2 孤立词建模	234
14.4.3 GMM-HMM	236
14.5 连续语音识别	239
14.6 Viterbi 解码	242
14.7 三音素状态聚类	244
14.8 判别式训练	247
参考文献	253
15 基于 WFST 的语音解码	255
15.1 有限状态机	256
15.2 WFST 及半环定义	256
15.2.1 WFST	256
15.2.2 半环 (Semiring)	257

15.3 自动机操作	259
15.3.1 自动机基本操作	260
15.3.2 转换器基本操作	261
15.3.3 优化操作	264
15.4 基于 WFST 的语音识别系统	276
15.4.1 声学模型 WFST	278
15.4.2 三音素 WFST	280
15.4.3 发音字典 WFST	280
15.4.4 语言模型 WFST	281
15.4.5 WFST 组合和优化	283
15.4.6 组合和优化实验	284
15.4.7 WFST 解码	285
参考文献	286
16 深度语音识别	287
16.1 CD-DNN-HMM	287
16.2 TDNN	291
16.3 CTC	294
16.4 EESEN	298
16.5 Deep Speech	300
16.6 Chain	309
参考文献	312
17 CTC 解码	314
17.1 序列标注	314
17.2 序列标注任务的解决办法	315
17.2.1 序列分类	315
17.2.2 分割分类	316
17.2.3 时序分类	317
17.3 隐马模型	317
17.4 CTC 基本定义	318
17.5 CTC 前向算法	320
17.6 CTC 后向算法	323
17.7 CTC 目标函数	324

17.8 CTC 解码基本原理	326
17.8.1 最大概率路径解码	326
17.8.2 前缀搜索解码	327
17.8.3 约束解码	328
参考文献	332
 第 4 部分 自然语言处理篇	 333
 18 自然语言处理简介	 334
18.1 NLP 的难点	334
18.2 NLP 的研究范围	335
 19 词性标注	 337
19.1 传统词性标注模型	337
19.2 基于神经网络的词性标注模型	339
19.3 基于 Bi-LSTM 的神经网络词性标注模型	341
参考文献	343
 20 依存句法分析	 344
20.1 背景	345
20.2 SyntaxNet 技术要点	347
20.2.1 Transition-based 系统	348
20.2.2 “模板化”技术	352
20.2.3 Beam Search	354
参考文献	356
 21 word2vec	 357
21.1 背景	358
21.1.1 词向量	358
21.1.2 统计语言模型	358
21.1.3 神经网络语言模型	361
21.1.4 Log-linear 模型	363
21.1.5 Log-bilinear 模型	364
21.1.6 层次化 Log-bilinear 模型	364

21.2 CBOW 模型	365
21.3 Skip-gram 模型	368
21.4 Hierarchical Softmax 与 Negative Sampling	370
21.5 fastText	371
21.6 GloVe	372
21.7 小结	373
参考文献	373
22 神经网络机器翻译	375
22.1 机器翻译简介	375
22.2 神经网络机器翻译基本模型	376
22.3 基于 Attention 的神经网络机器翻译	378
22.4 谷歌机器翻译系统 GNMT	380
22.5 基于卷积的机器翻译	381
22.6 小结	382
参考文献	383
 第 5 部分 深度学习研究篇	 385
23 Batch Normalization	386
23.1 前向与后向传播	387
23.1.1 前向传播	387
23.1.2 后向传播	391
23.2 有效性分析	392
23.2.1 内部协移	393
23.2.2 梯度流	393
23.3 使用与优化方法	394
23.4 小结	396
参考文献	396
 24 Attention	 397
24.1 从简单 RNN 到 RNN + Attention	398
24.2 Soft Attention 与 Hard Attention	398
24.3 Attention 的应用	399

24.4 小结	401
参考文献	402
25 多任务学习	403
25.1 背景	403
25.2 什么是多任务学习	404
25.3 多任务分类与其他分类概念的关系	406
25.3.1 二分类	406
25.3.2 多分类	407
25.3.3 多标签分类	407
25.3.4 相关关系	408
25.4 多任务学习如何发挥作用	409
25.4.1 提高泛化能力的潜在原因	409
25.4.2 多任务学习机制	410
25.4.3 后向传播多任务学习如何发现任务是相关的	411
25.5 多任务学习被广泛应用	412
25.5.1 使用未来预测现在	412
25.5.2 多种表示和度量	413
25.5.3 时间序列预测	413
25.5.4 使用不可操作特征	413
25.5.5 使用额外任务来聚焦	413
25.5.6 有序迁移	414
25.5.7 多个任务自然地出现	414
25.5.8 将输入变成输出	414
25.6 多任务深度学习应用	416
25.6.1 脸部特征点检测	416
25.6.2 DeepID2	417
25.6.3 Fast R-CNN	418
25.6.4 旋转人脸网络	419
25.6.5 实例感知语义分割的 MNC	421
25.7 小结	423
参考文献	424

26 模型压缩	426
26.1 模型压缩的必要性	426
26.2 较浅的网络	428
26.3 剪枝	428
26.4 参数共享	434
26.5 紧凑网络	437
26.6 二值网络	438
26.7 小结	442
参考文献	442
27 增强学习	445
27.1 什么是增强学习	445
27.2 增强学习的数学表达形式	448
27.2.1 MDP	449
27.2.2 策略函数	450
27.2.3 奖励与回报	450
27.2.4 价值函数	452
27.2.5 贝尔曼方程	453
27.2.6 最优策略性质	453
27.3 用动态规划法求解增强学习问题	454
27.3.1 Agent 的目标	454
27.3.2 策略评估	455
27.3.3 策略改进	456
27.3.4 策略迭代	457
27.3.5 策略迭代的例子	458
27.3.6 价值迭代	459
27.3.7 价值迭代的例子	461
27.3.8 策略函数和价值函数的关系	462
27.4 无模型算法	462
27.4.1 蒙特卡罗法	463
27.4.2 时序差分法	465
27.4.3 Q-Learning	466
27.5 Q-Learning 的例子	467

27.6 AlphaGo 原理剖析	469
27.6.1 围棋与机器博弈	469
27.6.2 Alpha-Beta 树	472
27.6.3 MCTS	473
27.6.4 UCT	476
27.6.5 AlphaGo 的训练策略	478
27.6.6 AlphaGo 的招式搜索算法	482
27.6.7 围棋的对称性	484
27.7 AlphaGo Zero	484
参考文献	484
28 GAN	486
28.1 生成模型	486
28.2 生成对抗模型的概念	488
28.3 GAN 实战	492
28.4 InfoGAN——探寻隐变量的内涵	493
28.5 Image-Image Translation	496
28.6 WGAN (Wasserstein GAN)	499
28.6.1 GAN 目标函数的弱点	500
28.6.2 Wasserstein 度量的优势	501
28.6.3 WGAN 的目标函数	504
参考文献	505
A 本书涉及的开源资源列表	506

第 1 部分

深度学习基础篇

1

概述

当今时代，人工智能（Artificial Intelligence, AI）、机器学习（Machine Learning, ML）、深度学习（Deep Learning, DL）都是耳熟能详的一些概念。机器学习是实现人工智能的一种方式，而深度学习是机器学习的一个分支。NVIDIA 的一张图（如图 1-1 所示）很好地概括了三者之间的关系^[1]。人工智能从 20 世纪 50 年代开始兴起，机器学习在 80 年代兴起，而深度学习的流行则晚一些，在 2010 年左右。

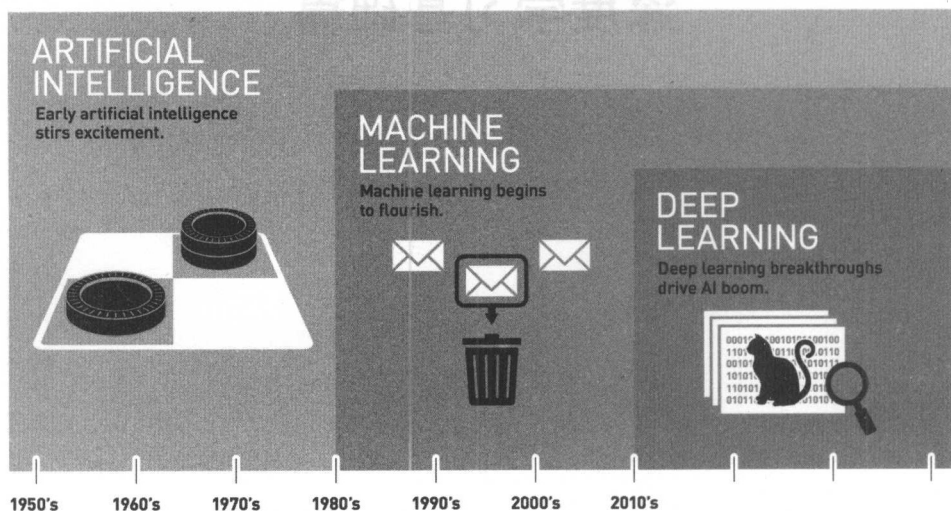


图 1-1 人工智能、机器学习、深度学习三者之间的关系

1.1 人工智能

2016 年 3 月 9 日至 15 日, Google 旗下 DeepMind 公司开发的围棋程序 AlphaGo 与世界围棋冠军、职业九段选手李世石进行人机大战, 最终以 4:1 的总比分赢得比赛。

2016 年年末至 2017 年年初, AlphaGo 在中国围棋网站上以 Master 账号与中、日、韩数十位围棋高手过招快棋, 连胜 60 局。

2017 年 5 月 23 日至 27 日, AlphaGo 迎战世界冠军柯洁, 以 3:0 的比分毫无悬念地赢下比赛。

2017 年 12 月 5 日, DeepMind 宣布, 新 AI AlphaZero 只须学习 34 小时即可战胜 AlphaGo。

随着 AlphaGo 的高歌猛进以及媒体的积极炒作, 人工智能的概念得到迅速普及, 甚至有人开始担心未来人工智能是否会危害到人类。

1.1.1 人工智能的分类

人工智能也称为机器智能, 是指人工制造出来的机器或系统展现出来的智能。

人工智能也可以进一步分为以下两种类型。

- 弱人工智能 (Weak AI): 通常是指机器通过机器学习之类的技术从大量数据中学到一些规律, 这种学习实际是记忆性的, 机器本身并无意识, 只是执行某些算法或任务的工具。弱人工智能有时也称为狭义人工智能 (Narrow AI)。
- 强人工智能 (Strong AI): 机器具有意识, 能够完全像人类一样思考和具有感情。强人工智能也称为通用人工智能 (General AI) 或全人工智能 (Full AI)。

AlphaGo 虽然看上去比人类还厉害, 但依然只是弱人工智能, 本身并无意识可言。而部分人担心的可能会危害到人类的人工智能, 则可以定义为第三类人工智能——超级智能 (Super-Intelligence) ——机器具有比人类更强大的智慧, 甚至是人类无法理解的智慧。

1.1.2 人工智能发展史

提及人工智能, 就不能不介绍计算机科学之父艾伦·麦席森·图灵 (Alan Mathison Turing, 1912—1954 年), 他在 1950 年创作的 *Computer Machinery and Intelligence* 中提出了智能的概念, 以及著名的图灵测试——计算机能否在智力行为上表现得和人没有区别。后来英国皇家学会规定的图灵测试标准为: 如果机器可以在 5 分钟内回答由人类测试者提出的一系列问

题，且其超过 30% 的回答可以让测试者认为是人类所回答的，则该机器通过图灵测试并认为其具有智能。时至今日，图灵测试一直被广泛用于测试机器是否具有智能。

人工智能比计算机出现得更早一些，最早出现在 1955 年的一次 10 人研讨会提案中，一般认为，1956 年的达特茅斯会议（Dartmouth Artificial Intelligence Conference）才是人工智能真正诞生的地方，当时参加会议的很多人都是大名鼎鼎的科学家，包括 John McCarthy（人工智能之父，1971 年图灵奖得主，Lisp 语言之父）、Marvin Minsky（人工智能之父，1969 年图灵奖得主）、Nathaniel Rochester（IBM 第一代通用计算机 701 主设计师）、Claude Shannon（信息论之父）。他们提出人工智能的研究目标是设计可以模拟人类的机器，这种机器可以使用语言，具有抽象理解能力。

至今，人工智能这个概念的提出已经半个多世纪了，为何很多人在最近一两年才真正切身感受到的存在？这一切与人工智能的发展历程密切相关。大体上，人工智能的发展可以分为以下几个阶段。

1. 第一个黄金时期（1956 年至 20 世纪 70 年代中）

1956 年达特茅斯会议后的 10 多年是人工智能发展的第一个黄金时期。在这个时期，大家认为逻辑推理能力是计算机具有智能的最重要原因，这个时期也被称为人工智能“推理期”。计算机被广泛用于解决代数题、证明几何定理等，这些成果得到了广泛赞赏，也让当时的研究者信心倍增，甚至很多人认为推理就是智能，有了推理能力就可以制造出完全智能的机器。

在这个时期，人工智能的相关研究也得到了政府的大力支持，获得了大笔的科研资金。1963 年 6 月，新建立的 ARPA（即后来的 DARPA, Defense Advanced Research Projects Agency, 美国国防部高级研究计划局）赞助了 MIT 222 万美元经费，用于资助 MAC 工程^[2]，其中包括 Marvin Minsky 和 John McCarthy 于 1959 年建立的人工智能实验室。此后 ARPA 每年提供 300 万美元人工智能研究经费，直到 20 世纪 70 年代。1963 年，John McCarthy 在斯坦福大学成立了另一个前瞻性的人工智能实验室。

2. 第一个低谷时期（20 世纪 70 年代中至 80 年代初）

虽然大家信心倍增，也投入了大量的科研资金，但是渐渐地发现当时的计算机运算能力有限，有限的内存和运算速度使得计算机很难处理实际应用中的人工智能问题，之前的盲目承诺无法兑现，光靠推理并没有实现真正的人工智能。人们渐渐失去耐心，批评和怀疑也接踵而至，1973 年有个著名的 Lighthill 报告（具体是指 James Lighthill 所写的论文 *Artificial Intelligence: A General Survey*），深度抨击了人工智能的进程，这导致之后英国的科研经费

大量转向其他方向。YouTube 上还有 1973 年 Lighthill 与 Richard Gregory、John McCarthy、Donald Michie 等人工智能支持者的辩论视频，感兴趣的读者可以搜一下 Lighthill debate。在一定程度上，Lighthill 报告加速了将人工智能打入冷宫的进程。从 1974 年开始，已经很难找到支持人工智能的科研经费。

在这个时期，感知机之类的联结主义遭遇冷落，1969 年 Minsky 和 Papert 出版了著作 *Perceptrons: an introduction to computational geometry*，书中暗示 1958 年由 Frank Rosenblatt（1928—1971 年）提出的感知机具有严重局限，从数学角度证明了单层感知机计算能力有限的根本原因，指出单层感知机甚至连异或（XOR）这样的问题也不能解决，并论证了单层感知机的这些局限性在多层感知机中是不可能被全部克服的。此处详情可参考本书 2.3.2 节。

同样在这个时期，专家系统之父 Edward Albert Feigenbaum（1994 年图灵奖得主）等研究者开始倡导智能机器必须具备知识，据此可以认为，20 世纪 70 年中后期人工智能进入了“知识期”。在这个时期，专家系统代替逻辑推理成为新宠，人们总结出来的大量知识通过规则之类的系统输入计算机，规则的详细程度决定了机器的智能水平，现在人们常说的“有多少人工就有多少智能”是非常适合这个时期的。

3. 第二个黄金时期（20 世纪 80 年代初至 80 年代末）

20 世纪 80 年代，人工智能开始复苏。1981 年，日本经济产业省为支持第五代计算机项目拨款 8.5 亿美元，这种计算机可以与人对话、翻译、理解图像，并能像人一样推理。此后，英国、美国纷纷效仿，也启动了很多相关项目，人工智能迎来了新的发展时期。

20 世纪 80 年代初，另一个令人振奋的事件是 John Hopfield 和 David Rumelhart 使联结主义重获新生。1982 年美国加州理工学院物理学家 John Hopfield 博士提出了 Hopfield 网络，这是一种递归神经网络，从输出到输入增加了反馈连接；1986 年 David Rumelhart 等人发表了 *Parallel Distributed Processing*，文中详细讲解了具有非线性连续变换函数的多层感知机的误差反向传播（Error Back Propagation, BP）算法，时至今日的深度学习，BP 依然是中流砥柱。

4. 第二个低谷时期（20 世纪 80 年代末至 90 年代初）

然而，好景不常在，专家系统慢慢暴露出维护难、不完善等缺点。1987 年，Apple 和 IBM 的普通台式机的性能反而超过了“智能计算机”，专家系统被质疑。20 世纪 80 年代末，DARPA 对人工智能的期望也降低了。1991 年，日本的第五代计算机宣告失败，人工智能再一次跌入低谷。

5. 第三个黄金时期（20 世纪 90 年代中至今）

从 20 世纪 90 年代中期开始，人工智能陆续走向台前，处于第三个黄金发展时期，在这个时期发生的事件包括：

- 1997 年，深蓝击败国际象棋世界冠军卡斯帕罗夫。
- 2011 年，IBM Watson 在美国电视知识抢答竞赛节目“危险边缘 (Jeopardy!)”中击败了史上胜率最高的两位人类冠军。
- 2012 年至今，ImageNet 图像分类大赛年年创新高，并在一定程度上超过了人类的识别能力。
- 2016—2017 年，Google DeepMind 创造的 AlphaGo 围棋系统相继战胜世界冠军李世石、柯洁。

世界大国纷纷布局人工智能，美国在 2013 年 4 月由奥巴马政府宣布投入 1.1 亿美元，并从 2014 年开始，每年各投入 3~5 亿美元，十年总计将投入 45 亿美元。

欧盟在 2013 年年初公布了总投资 12 亿欧元的十年计划，预计在 2018 年之前开发出一个具有意识的智能大脑，同时在 2014 年 6 月启动了机器人研发计划，目标是为欧盟各行各业提供机器人。

日本在 2015 年 12 月开展了第五个科学与技术基础五年计划，预计总投资 26 万亿日元，用来实现一个全球领先的“超级智能社会 (Super)”，以及发展信息技术以及人工智能、机器人等相关技术。

韩国在 2013 年 5 月提出了 ExoBrain 十年计划，预计总投资 9000 万美元，计划开发专业领域的人机对话系统。

中国在 2015 年 7 月发布了《国务院关于积极推进“互联网+”行动的指导意见》，其中人工智能是重点布局的 11 个领域之一；2016 年 5 月发改委公开了《“互联网+”人工智能三年行动实施方案》；同时，脑科学研究也上升到国家战略高度。

而在这一次人工智能的黄金发展过程中，深度学习起到了至关重要的作用，除上面提到的 ImageNet 图像分类大赛、AlphaGo 与深度学习密切相关外，深度学习使研究者在计算机视觉、语音识别、机器翻译等各个领域都取得了有史以来的最长足进步。深度学习依然是从古老的联结主义发展而来的，是隐层多于一层的神经网络。之所以在这个时期深度学习才得到大力发展，主要原因包括三点：

- 算法——逐层训练初始化模型、分布式并行训练算法等能力的提升。
- 计算——GPU (Graphics Processing Unit)、FPGA (Field-Programmable Gate Array)、TPU (Tensor Processing Unit) 等能力的大幅提升。

- 大量的训练数据——例如 ImageNet 千万级别的图像数据。

当然，在这个时期也存在泡沫，有人鼓吹传统行业的从业者即将失业，有人担心自己要被机器取代甚至消灭，甚至很小的深度学习创新也开始被媒体捧到天上去，生怕大家觉得这个创新的影响力不够大。作为技术人来说，刨去这些泡沫，深度学习在很多领域都带来了有史以来最大的技术突破，甚至很多突破远远超出了技术实践者本身的预期。

1.2 机器学习

机器学习（Machine Learning）是一门研究计算机模拟和实现人类行为的科学，通过不断改善知识结构，进而超越人类能力的学科。机器学习算法是从数据中自动分析并获得规律，进而可以对未知数据进行预测的算法。

1.2.1 机器学习的由来

很多时候，人们希望能借助机器的力量来自动完成一些任务，从而将人类从烦琐的事项中解放出来。比如自动监测违规车辆及排查嫌疑车，可以代替交通警察用人眼监控显示屏；自动驾驶，可以选择最佳路线、躲避其他车辆而安全地驾驶；自动人脸识别，可以代替人工完成特定的服务。

概括来说，这个过程涉及了两大步骤。

- 认知这个世界，获取信息。
- 根据信息进行判断和决策。

从人类的角度看，第2步显然重要得多。人们进行了种种努力，不断探索如何能排除感性的干扰，做出更加理性的决策。这样的决策在给定信息的情况下，被称为“全局最优”策略。而这一步，在机器看来却轻巧得多，它可以充分调动强大的计算能力，综合各种优化算法，在极短的时间内就能给出最优的答案。

但另一方面——“像人类一样认知这个世界”——却不是它的强项。面对一张图片，它可以告诉你一共有多少个像素点，也可以准确地给出图片上每一个像素点的像素值，但却分辨不出那些像素点组成的脸庞。如图 1-2 所示，十年前机器学习领域还在聚焦于如何能让机器准确地识别类似简单的物体。机器的识别能力甚至比不上一个 3 岁的孩童。不仅如此，在经过训练能认出图片上的物体后，一旦光影变幻、物体遮挡或角度变化，就很可能又会识别失败。



图 1-2 Caltech 101 数据集

人们挠挠头，不知道怎么教机器这个憨憨的学生去“感知”这个世界。于是人们转而看看关于自己大脑的研究，也就是神经科学，希望能获得一些关于“认知”的理论。可惜，当时大脑神经科学也是一片广阔而充满了未解之谜的领域。虽然对神经元的研究、激活和信息传导有了一定的成果，但还不足以解释“认知”这个宏大的课题。尽管如此，人们还是乐观地开始了对机器学习领域中的人工神经网络的研究。经过大半个世纪的坎坷和沉浮，厚积薄发，在 21 世纪开始大放异彩，在各个领域都取得了惊人的进展。

机器学习的领域很广泛，与视觉相关的领域包括：物体识别、图像分割、图像索引、人脸识别、场景识别、场景匹配等。而且还有很多有趣的商业应用，比如谷歌眼镜等。

与听觉相关的领域包括：语音识别、乐曲片段匹配，甚至有自动作曲这样有趣的方向。

与认知相关的领域包括：自然语言处理、专家系统等。

基于对人群偏好的推测而进行的内容推荐，也由于有着广阔的应用场景而成为机器学习中很热门的一个领域，包括诸如网页推荐、广告推荐、购物产品推荐、电影推荐等。

此外，还有很多其他五花八门的方向，比如机器人的相关研究。如图 1-3 所示，斯坦福的 Jackrabbot 就是一个带着领带、风度翩翩的社会化行走机器人。和其他机器人不同，它在人行道上行走时，会特别学习人类的社会习惯，比如行走时注意他人的个人空间、有礼貌地行走，而不是只为了走到目的地而加快步伐地横冲直撞。

这里，我们简单澄清一些与机器学习密切相关且容易混淆的概念。

模式识别 (Pattern Recognition) ——在一定程度上等同于机器学习，一般认为模式识别最初来自于工业界，而机器学习来自于学术界，机器学习的经典书籍 *Pattern Recognition and Machine Learning* 所讲的就是两者不分家。

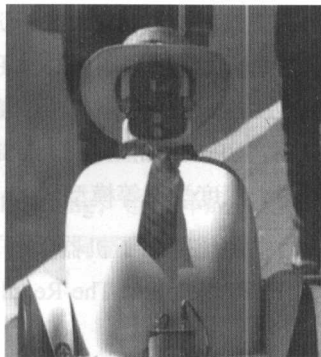


图 1-3 Jackrabbot 机器人会遵从社会习俗

统计学习 (Statistical Learning) ——也是机器学习的近义词, 机器学习的很多方法都源自于统计学习, 这些方法往往具有优美的数学推导, 比如支持向量机 (SVM) 方法等。总体上, 统计学习更偏数学理论一些, 而机器学习更偏实践一些。

数据挖掘 (Data Mining) ——在有些场景中也被等同于机器学习, 但更专业的解释应该是机器学习在大数据领域的应用, 通过机器学习的方法从大数据中挖掘出规律或知识。

计算机视觉 (Computer Vision) ——强调机器学习在图像领域的应用。可以说, 迄今为止, 计算机视觉是机器学习尤其是深度学习最成功应用的领域, 没有之一。

语音识别 (Speech Recognition) ——研究机器听懂人类声音的领域。目前语音识别也取得了长足的进步, 有 Siri、语音输入法等大家耳熟能详的应用。

自然语言处理 (Natural Language Processing) ——研究机器理解人类语言的领域。相比计算机视觉、语音识别的感知问题, 自然语言处理尤其是其中的语义理解属于认知问题, 相对更难一些。

那么具体来说, 机器学习是什么呢? 它和本书要讲的神经网络以及深度学习是什么关系呢? 下面我们将探讨这些问题。

1.2.2 机器学习发展史

由于机器学习只是实现人工智能的一种方式, 所以人工智能的发展史实质上包括了机器学习的发展历程。“推理期”“知识期”“学习期”就是指与机器学习相关的主流时期, 感知机、支持向量机、神经网络等又是机器学习具体的模型, 在此不再赘述。

可以说, 1996 年至今, 机器学习在工业界得到广泛应用, 从而使机器学习的发展达到一个前所未有的新高度。

比如搜索引擎中的分词、新词挖掘、垃圾网页过滤、网页滤重、Learning to Rank、PageRank、主题模型、摘要提取、特征学习等，大量使用了机器学习中的逻辑回归（Logistic Regression, LR）、支持向量机（Supported Vector Machine, SVM）、GBDT（Gradient Boosting Decision Tree）、Latent Semantic Analysis（LSA）/Probabilistic Latent Semantic Analysis（PLSA）/Latent Dirichlet Allocation（LDA）、概率图、深度学习等模型。

再比如在计算广告点击率预测中广泛使用了机器学习中的 LR（Logistic Regression）、BPR（Bayesian Probit Regression）、FTRL（Follow-The-Regularized-Leader）、Online Learning、深度学习等相关技术。

而计算机视觉、语音识别、机器翻译等更是在近几年被深度学习不断刷新高度。

1.2.3 机器学习方法分类

机器学习方法可以大致分为监督学习、无监督学习、半监督学习、增强学习等几类。

监督学习（Supervised Learning）——通过对标注的训练数据进行学习，得到一个从输入特征到标签的映射模型，再利用这个模型对未知标签的新数据进行预测。比如我们拥有大量正常内容的邮件，同时拥有大量垃圾邮件，那么就可以训练一个监督学习模型来做垃圾邮件分类，最终得到的模型就能鉴定新邮件是否是垃圾邮件。

监督学习又可以进一步分为分类（Classification）和回归（Regression）等类别。如果标签是离散类别的，则一般认为是分类问题，比如前面提到的垃圾邮件分类等；而如果标签是连续数值型的，则一般认为是回归问题，比如房价的预测问题等。

无监督学习（Unsupervised Learning）——不需要对训练数据进行标注，直接对数据进行建模。比如一堆杂乱无章的文字片段或者图片，我们完全可以根据文字或图片本身的内容对其进行大致的归类。

无监督学习比较常见的类别有聚类（Clustering）、密度估计（Density Estimation）和降维（Dimension Reduction）等。其中，聚类是根据样本之间的特征相似度将一组数据聚为一类，使得类内的数据相似度比不同类间的数据相似度更高。密度估计是根据数据集统计推断样本集对应的概率分布。降维，顾名思义，就是降低输入数据的维度。在很多应用中，原始数据具有非常高的维度（比如在广告点击率预测应用中，特征维度往往达到上亿级别），而且有很多特征是冗余或者不相关的，降维算法有助于去除无关特征、合并冗余特征。

半监督学习（Semi-Supervised Learning）——介于监督学习和无监督学习之间的方法。在实际应用中，数据标注往往对模型的学习非常有帮助，但代价也不低，有时候甚至超过了可以忍受的限度，这时候半监督学习就是一种很好的选择。半监督学习的方法非常多，其中

滚雪球式的主动学习（Active Learning）是数据挖掘中非常常用的方法，利用学习算法主动选出最值得标注的数据进行人工标注，标注完成后，新的标注数据和之前的标注数据合在一起继续进行训练，训练完毕后继续用算法甄选性价比最高的数据进行人工标注，如此不断迭代，最后得到的模型效果往往非常好。

增强学习（Reinforcement Learning，也翻译成强化学习）——一种交互式的学习方法，模型根据环境给予的奖励或惩罚不断调整自己的策略，尽量获得最大的长远收益。相关的具体介绍可以参考第 27 章。

1.2.4 机器学习中的基本概念

在机器学习算法中，目前在业界得到较多应用的主要是监督学习，监督学习需要训练数据，其本身由模型、策略和算法^[3]三要素组成。

机器学习的模型是从数据中学到的用来描述数据所在空间的数学模型，可以说是经过了数学抽象的规律。模型是机器学习的最终目的，有了模型，才能对未知数据进行预测或分析。在监督学习中，模型一般指具体的条件概率分布模型或者决策模型。模型的假设空间 \mathcal{F} 是所有可能的条件概率分布或者决策函数。这里为了简单起见，仅以决策函数为例进行说明。假设输入集合为 X ，对应的输出集合（标签，Label）为 Y ，其假设空间可以表示为：

$$\mathcal{F} = \{f|Y \sim f(X)\}$$

机器学习常用的模型有很多，比如线性模型、逻辑回归、Softmax、神经网络/深度学习、SVM、决策树、随机森林、GBDT、与矩阵分解相关的系列模型等。

由于假设空间对应的函数有很多，对于如何选择就需要引入特定的评估策略，机器学习一般引入损失函数（Loss Function）或代价函数（Cost Function）来评估预测错误的程度。

常见的损失函数如表 1-1 所示，其中 y 表示对应输入样本 x 的 Label， \hat{y} 为函数预测值 $f(x)$ ，注意这里只针对单个样本计算损失。

表 1-1 常见的损失函数

损失函数	公式
0-1 损失	$L(y, \hat{y}) = \begin{cases} 0, & y = \hat{y} \\ 1, & y \neq \hat{y} \end{cases}$
绝对值损失	$L(y, \hat{y}) = y - \hat{y} $

续表

损失函数	公式
平方差损失	$L(y, \hat{y}) = (y - \hat{y})^2$
负 Log 似然	$L = -y \log p(y = 1 x) - (1 - y) \log(1 - p(y = 1 x))$

损失函数在输入输出联合概率分布 $p(x, y)$ 下的期望称为风险函数 (Risk Function) 或者期望损失 (Expected Loss), 用 $R(f)$ 表示^[3]。

$$R_{\text{exp}}(f) = E_p[L(y, \hat{y})] = \int L(y, \hat{y})p(x, y)dxdy$$

机器学习的目标是要选择一个期望损失最小的模型, 但是这在现实中不可行, 因为联合概率分布 $p(x, y)$ 表征的是所有样本遵循的分布, 一般无法求得。只能退而求其次利用训练数据集的平均损失近似表示, 这个平均损失称为经验风险 (Empirical Risk) 或者经验损失 (Empirical Loss), 记为:

$$R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^N L(y_i, \hat{y}_i)$$

其中, N 为样本数量。根据大数定律, 当 N 趋近于无穷大时, $R_{\text{emp}}(f)$ 趋近于 $R_{\text{exp}}(f)$ ^[3]。

算法是机器学习的具体学习方法, 也可以称为优化算法, 包括: 梯度下降法、牛顿法、拟牛顿法等^[4]。

1.3 神经网络

在人工智能的发展史中, 我们已经反复发现了神经网络的踪影。作为机器学习的一个分支, 神经网络的发展也是跌宕起伏的。

神经网络 (Neural Network) 是从人类脑神经元的研究中获得灵感, 模拟其神经元的功能和网络结构, 来完成认知任务的一类机器学习算法; 还有一类机器学习算法, 则不局限于神经元, 而是尝试将问题从数学上抽象, 从而对该简化的数学问题进行研究并做出解答。

而深度学习 (Deep Learning), 则是指多层神经网络, 即隐层大于一层的神经网络。在后面的章节中, 我们还会详细地讲一讲网络结构和隐层到底是什么。

1.3.1 神经网络发展史

1. 神经网络的提出与发展（1943—1969 年）

早在 1943 年，人工神经网络就已由 McCulloch 和 Pitts 提出，他们分析了理想化的人工神经元网络，并且指出了它们运行简单逻辑运算的机制。但这仅仅是一种理想化的蓝图。直至将近 15 年后，康奈尔大学的实验心理学家 Frank Rosenblatt 在一台 IBM-704 计算机上模拟实现了一种他发明的叫作“感知机”的神经网络模型，人工神经网络才走进了现实。稍后，伴随着 Frank Rosenblatt 出版的一本名为《神经动力学原理：感知机和大脑机制的理论》的书，感知机迅速获得了人们的关注，并被寄予了极高的期望。

然而，1969 年，一本名为《感知机》的书详细地分析了感知机的适用范围，并明确提出对于简单的异或逻辑问题，感知机都由于其非线性而无法解决，而现实中的问题恰巧大多都不是线性可分的。尽管在 5 年后，Werbos 的博士论文证明了，只要在感知机的网络中多加一层，并且利用“后向传播”的学习方法，就可以解决异或问题，但是人们依然对感知机持悲观的态度。不仅如此，这种看法还扩大到所有的神经网络科学上，以至于对整个神经网络的研究陷入了停滞状态。

为了本书内容的简洁性，以下如不特指，“神经网络”均指代“人工神经网络”。

2. 神经网络的困境与 SVM 的独领风骚（1971—2005 年）

从 20 世纪 70 年代开始，人们对神经网络的研究热情不断下降。与此同时，以 Vapnik 为首的科学家创造性地提出了 VC 维的概念，以及结构风险最小化原则。Vapnik 是研究统计学出身的，数学功底深厚。随着这个理论的深入，并经过 20 年的摸索后，Cortes 和 Vapnik 等人在 1993 年提出了“支持向量机”（Support Vector Machine），成功地将其应用于实际问题中。支持向量机旨在利用核（Kernel）技巧把非线性问题转换成线性问题，解决了感知机所不能解决的问题，一时间独领风骚。其坚实的理论基础和解决现实问题的有效性，使它获得了广泛的认可。而同时，Vapnik 等统计机器学习理论专家从理论的角度怀疑神经网络的泛化能力，学术界对于神经网络的研究也更加趋于悲观。

尽管如此，在这长达半个世纪的冰河期，依然有神经网络学家在坚守着自己的阵地。1982 年，Hopfield 提出了一种新的神经网络，它可以解决一大类模式识别问题，并且可以给出一类组合优化问题的近似解。1986 年，Rummelhart 与 McClelland 再次提出了神经网络的学习算法——后向传播。LeCun 也发明了卷积神经网络，并利用其实现自动提取图像的特征，成功地完成了手写数字的识别。这些都为后来神经网络的再次兴起奠定了坚实的基础。

3. Hinton 引领的神经网络复兴（2006 年）

2006 年，Hinton 提出了深度神经网络（深度学习）。深度神经网络指的是隐层大于一层的网络结构。Hinton 提出首先用 Restricted Boltzmann Machine 经过非监督学习来学习出网络结构，然后再由后向传播算法学习网络内部的参数值。

尽管如此，深度学习仍然广受质疑。于是 Hinton 带领其学生埋头苦干，于 2012 年在计算机视觉领域的著名比赛——ImageNet 分类比赛中，以高出第二名 10 个百分点的战绩高地夺得第一名。这是深度学习在沉寂半个世纪后，第一次在机器学习领域的比赛中参赛，并且取得了卓著的成绩，震动了整个机器学习界。这一成绩坚实地印证了深度学习的有效性，并随后在各个领域也都迅速拔得头筹。深度学习正式进入了复兴和辉煌的时代。

这一次复兴，离不开 Hinton、LeCun、Bengio（关系如图 1-4 所示）和其他优秀研究者（比如第四巨头 Andrew Ng 等）的努力工作，他们坚信神经网络的有效实用性，并不断摸索真正可行的神经网络道路。21 世纪不断普及的大数据以及高度并行的计算设备——图形处理单元（Graphics Processing Unit, GPU）也为神经网络提供了必不可少的支持。有了这些，才有了深度学习在各个领域遍地开花的今天。

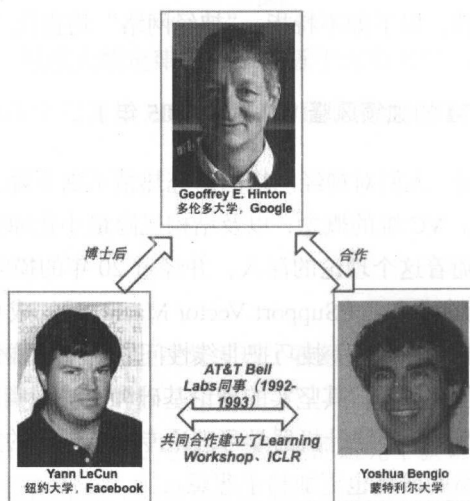


图 1-4 深度学习巨头 Hinton、LeCun 和 Bengio 的关系

回顾神经网络的兴起——衰落——复兴乃至辉煌的过程，不禁让人唏嘘。如今深度学习研究的大放异彩，离不开大师们近半个世纪的坚守和在质疑中坚定地前行。这不仅需要灵感，还需要魄力，以及一以贯之的坚定的信心。

深度学习在图像识别领域大获成功之后，又被迅速应用到其他问题上。看起来各不相同的问题，一旦理解它们仅仅是特征不同、基于特征都要完成对应的分类问题时，各个问题似乎就有了相似之处。当然，在实践中，能成功地把深度学习应用于各类问题上还是需要相当的想象力、创造力以及对模型的把控力的。图 1-5 列举了部分精彩的实例，有些似乎超出了人们的想象，却都成为了现实。而对于实现这些有趣应用的神秘而强大的深度学习，我们也将揭开它的面纱。



图 1-5 深度学习在各领域遍地开花

参考文献

- [1] Deep Learning Explained - NVIDIA. https://www.nvidia.com/content/dam/en-zz/Solutions/deep-learning/home/DeepLearning_eBook_FINAL.pdf.
- [2] MAC 工程. <http://www.multicians.org/project-mac.html>.
- [3] 李航. 统计学习方法. 北京: 清华大学出版社, 2012.
- [4] Stephen Boyd and Lieven Vandenberghe. Convex Optimization. Cambridge University Press.



2

神经网络

深度学习网络是指隐层大于一层的神经网络，所以了解神经网络的基本原理是学习深度学习必不可少的步骤。

本章首先讨论生物神经元的相关研究，然后介绍常用的神经元，最后以感知机和深度神经网络为例介绍神经网络的基本原理。

2.1 在神经科学中对生物神经元的研究

神经元相互连接组成神经网络。每一个神经元从其他神经元处获得输入信息，少部分神经元也从接收器获得信息；神经元处理这些输入信息，一旦被激活，就会继续发送信号至其他相连的神经元。

机器学习中的神经元以生物神经元为原型，受到了其机制不少的启发和影响；然而，为了可以顺利地完成模型的实现，不可避免的，对机器学习中的神经元进行了不少抽象和简化，甚至有些已经跳出了生物神经元的束缚。下面我们首先了解生物神经元的机制，然后再详细了解机器神经网络的神经元模型。

2.1.1 神经元激活机制

神经生物学家 David Hubel 和 Torsten Wiesel 由于发现了“视觉系统的信息处理”而荣获 1981 年的诺贝尔生理或医学奖。1958 年，他们通过实验证实了位于后脑皮层的神经元与视

觉刺激之间存在某种对应关系。换句话说，一旦视觉受到了某种刺激，后脑皮层的特定部分的神经元就会被激活。他们的实验发现了一种被称为“方向选择性细胞”的神经元，当看到眼前物体的边缘，而且这个边缘指向某一个方向时，这种神经元就会被激活。

神经生物学家认识到，生物神经元是神经系统的重要组成单位之一。随后的深入研究揭示了神经元的基本构造由细胞体和神经突（包括树突、轴突、突触）组成，如图 2-1 所示。树突呈树状分支，为神经元的“信息接收区”，它将受到刺激引起的电位变化向胞体传递；然后会有一个“触发区”负责整合电位，决定是否达到阈值，从而产生神经冲动；细长的轴突为“传导区”，而其末端的突触为“输出区”——神经冲动会导致突触释放出神经传递物质或者电力，从而实现将整合的信息向下一个神经元进行传递的过程。

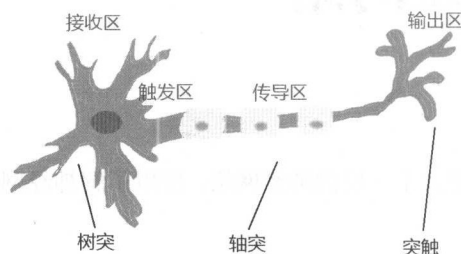


图 2-1 生物神经元模型

机器学习中的神经元也暗合了这几个功能区域：接收区、触发区、传导区、输出区。其中，触发区最重要，不同的触发机制也标志着不同类型的神经元。

2.1.2 神经元的特点

除神经元的基本激活机制以外，科学家发现，大脑不同位置的神经元似乎专门实现各自的功能。尽管如此，但是各种神经元本身的构成却很相似。在研究中甚至发现，早期的大脑损伤，其功能可能是以其他部位的神经元来代替实现的。当然，在生物体中，这需要在非常早期才有可能。有趣的是，在深度学习中也有类似的实现：在一个数据集上训练成型的深度神经网络，在另一个完全不同的数据集上只需稍加训练，就有可能适应和完成那个新的任务。这在机器学习被称为“迁移学习”（Transfer Learning）。

此外，科学家还发现，神经元具有稀疏激活性，即尽管大脑具有多达五百万亿个神经元，但真正同时被激活的仅有 1%~4%。这种稀疏激活性也影响了机器学习中的神经元的模型设计，比如稍后提到的 ReLU 神经元，对小于 0 的输入都进行了抑制，极大地提高了选择性激活的特征。在 Dropout 及其他剪连接策略中，稀疏性也得到应用。

上面简单了解了生物神经元的机制后，下面来具体了解机器学习中的神经元的代表模型。

2.2 神经元模型

前文提到，生物神经元被以多种形式抽象和简化，但它们都有一个共同的特征，即由输入、激活函数、输出构成。各种神经元的简化模型的不同之处就在于激活函数不一样。图 2-2 列出了几种基本神经元。

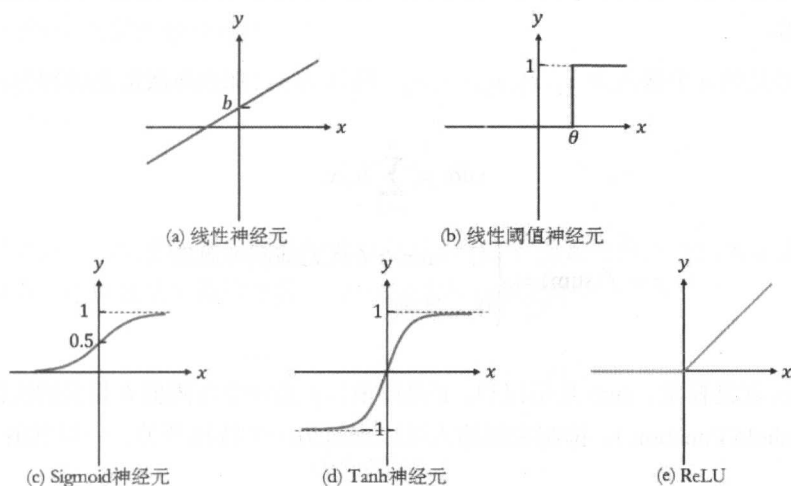


图 2-2 神经元的简化模型

2.2.1 线性神经元

线性神经元 (Linear Neuron) 是指输出与输入呈线性关系的一种简单模型。其表达式为 $y = wx + b$ 。如图 2-2 (a) 所示，它实现的是输入信息的完全传导。在现实中，由于其缺乏对信息的整合而基本不被使用，仅作为一个概念基础。

2.2.2 线性阈值神经元

早在 1943 年，人工神经网络 (Artificial Neural Network, ANN) 的提出者 Warren Sturgis McCulloch (1898—1969 年) 和 Walter Harry Pitts (1923—1969 年) 就分析了一种简单的人

工神经元模型，并且指出了它们运行简单逻辑运算的机制。这种简单的神经元采用线性神经元和二值“开/关”相结合，称为线性阈值神经元（Linear Threshold Neuron），也被称为 McCulloch-Pitts 神经元。它具有以下特征。

- 输入和输出都是二值的。
- 每个神经元都具有一个固定的阈值 θ 。
- 每个神经元都从带有权重的激活突触接收输入信息。
- 抑制突触对任意激活突触有绝对否决权。
- 每次汇总带权突触的和，如果大于阈值 θ 而且不存在抑制突触输入，则输出为 1，否则为 0。

假定神经元的 n 个输入为 $x_1, x_2, x_3, \dots, x_n$ ，输出为 y ，那么每次汇总的和为：

$$\text{sum} = \sum_{i=1}^n w_i x_i$$

$$y = f(\text{sum}) = \begin{cases} 1, & \text{sum} \geq \theta \text{ 且无抑制突触输入} \\ 0, & \text{其他} \end{cases}$$

其中， w_i 就是权重，sum 是带权和， θ 是阈值， f 是一个与阈值 θ 相关的线性阈值函数（Linear Threshold Function）。抑制突触输入可以理解为一个特权开关，一旦其值为 1，则输出必为 0。

以函数 $y = \bar{x}_1 x_2 + x_2 \bar{x}_3$ 为例，其中 x_1, x_2, x_3 是布尔输入， y 是上帝知道的真实标注（Label）， \hat{y} 是神经元的输出。现假定权重向量 $\mathbf{w} = [-1, 2, -1]$ ，阈值 θ 为 $\frac{1}{2}$ ，且没有抑制突触输入。由之前定义可知， $\text{sum} = \sum_{i=1}^n w_i x_i = -x_1 + 2x_2 - x_3$ 。考虑 x_1, x_2, x_3 的 8 种不同取值，我们可以得到如表 2-1 所示的相关结果。可以看出，这个神经元能完美模拟这个布尔函数。

表 2-1 布尔函数 $y = \bar{x}_1 x_2 + x_2 \bar{x}_3$ 的“开/关”神经元计算表

x_1	x_2	x_3	sum	\hat{y}	y
0	0	0	0	0	0
0	0	1	-1	0	0
0	1	0	2	1	1
0	1	1	1	1	1
1	0	0	-1	0	0
1	0	1	-2	0	0

续表

x_1	x_2	x_3	sum	\hat{y}	y
1	1	0	1	1	1
1	1	1	0	0	0

2.2.3 Sigmoid 神经元

Sigmoid 神经元可以使输出平滑而连续地限制在 0~1 的范围内，它靠近 0 的区域接近于线性，而远离 0 的区域为非线性。Sigmoid 神经元可以将实数“压缩”至 0~1 的范围内，大的负数趋向于 0，大的正数则趋向于 1。

Sigmoid 神经元的数学表达式为：

$$y = \frac{1}{1 + e^{-x}}$$

虽然它的激活函数看起来比前面的模型要复杂不少，但是它的求导结果很漂亮（在训练神经网络中需要计算激活函数的导数）。具体的求导运算如下：

$$\begin{aligned}
 \frac{\partial y}{\partial x} &= -\frac{1}{(1 + e^{-x})^2} \cdot e^{-x} \cdot (-1) \\
 &= \frac{e^{-x}}{(1 + e^{-x})^2} \\
 &= \frac{1}{1 + e^{-x}} \cdot \frac{1 + e^{-x} - 1}{1 + e^{-x}} \\
 &= y \cdot (1 - y)
 \end{aligned}$$

由此可见，Sigmoid 的导数可以直接用它的输出值来计算，非常简单。

Sigmoid 函数在过去被广泛使用，除求导简单外，还源于它很好地阐释了一个神经元的“燃烧率（Firing Rate）”：从一个假定的完全不激活（0）到完全饱和的燃烧（1）。

但 Sigmoid 神经元近年来变得鲜少使用。它的两个主要缺陷如下：

（1）Sigmoid 函数进入饱和区后会造成梯度消失

Sigmoid 神经元的—个非常不受欢迎的属性是在函数两端响应趋向于饱和（接近于 0 或 1），这些区域的梯度几近于 0。在后向传播中，这个（局部）梯度将以乘数的关系进入整个优化过程。这样，如果局部梯度值很小，将很有效地“杀掉”梯度，使得几乎没有信号流过神经元到达它的权重并递归回数据。此外，在初始化 Sigmoid 神经元参数时也需要加倍小心，

以避免函数进入饱和区。例如，如果初始化的参数值过大，大部分神经元工作在饱和区，则网络变得很难学习。

(2) Sigmoid 函数并非以 0 为中心

这一属性同样不受欢迎，因为通过神经元向后传播的网络需要处理非 0 的数据，这将对梯度下降的过程造成影响。因为如果进入一个神经元的数据总是正的（如 $f = w^T x + b, x > 0$ ），那么在反向传播时参数 w 的梯度要么都是正的，要么都是负的（取决于整个表达式 f 的符号）。这可能导致更新参数 w 时出现恼人的 zig-zag 运动。不过，当这些梯度在一个批处理数据中先进行累加，最后再更新参数时，符号可能会发生变化，这在一定程度上可以降低影响。

2.2.4 Tanh 神经元

Tanh 神经元是 Sigmoid 神经网络的一个继承，它将实数“压缩”至 $-1 \sim 1$ 的范围内，因此改进了 Sigmoid 变化过于平缓的问题。

Tanh 神经元的数学表达式为：

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Tanh 的求导结果如下：

$$\frac{\partial y}{\partial x} = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} = 1 - y^2$$

2.2.5 ReLU

整流线性单元（Rectified Linear Unit），又称为修正线性单元，一般以其英文缩写 ReLU 来指代。其数学表达式为：

$$y = \begin{cases} x, & x > 0 \\ 0, & \text{其他} \end{cases}$$

该函数等价于 $y = \max(0, x)$ 。它在阈值以下的输出都被截断成“0”，在阈值以上的输出则线性不变，如图 2-3 左图所示。Krizhevsky 等的实验表明 ReLU 比 Tanh 的收敛速度快 6 倍^[1]，如图 2-3 右图所示。

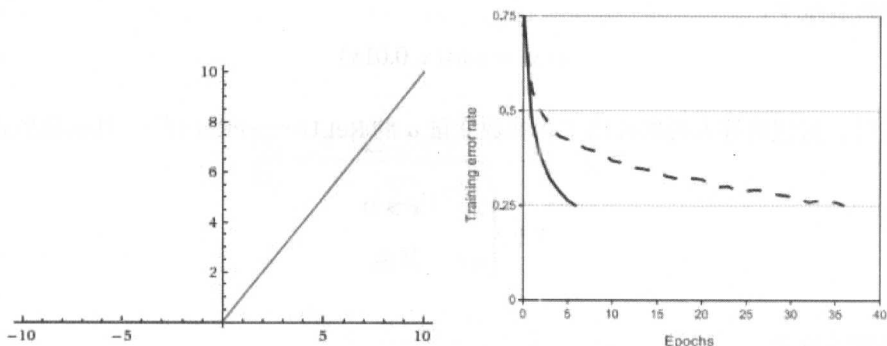


图 2-3 ReLU 函数图像（左）及 ReLU 与 Tanh 的实验对比（右）

ReLU 是分段可导的，其导数形式非常简单：

$$\frac{\partial y}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & \text{其他} \end{cases}$$

ReLU 在神经网络的实际应用中被广泛采用，因为其既具有非线性特点，使得信息整合能力大大增强；在一定范围内又具有线性特点，使得其训练简单、快速。使用 ReLU 有以下优点。

- 相比 Sigmoid 和 Tanh，ReLU 在随机梯度下降过程中能够明显加快收敛速度（有人认为这是由于它具有线性、非饱和的形态）。
- 相比 Sigmoid 和 Tanh 包含复杂算子，ReLU 通过简单的阈值操作就能实现。

然而，ReLU 并不是万能的，在训练过程中可能是脆弱的并且出现“死亡”。例如，流经 ReLU 神经元的大梯度可能导致权重更新到不再被任何数据激活的位置上。如果发生这种情况，流经该神经元的梯度将永远为 0。也就是说，在训练过程中，ReLU 单元会不可逆地死去。如果学习率设得太高，那么在网络中甚至有高达 40% 的神经元不能被激活。通过调整学习率，能够限制这种情况的发生。

针对 ReLU 的相关缺点，近年来又出现了很多变种，包括 Leaky ReLU^[2] 试图解决 ReLU “死亡”单元的问题。当 $x < 0$ 时，函数不再直接取 0，而是取 $0.01x$ ，即：

$$y = \begin{cases} x, & x > 0 \\ 0.01x, & \text{其他} \end{cases}$$

该函数等价于：

$$f(x) = \max(x, 0.01x)$$

再往后，何恺明等人再次提出了含参数变量 α 的 ReLU——PReLU^[3]，其函数形式为：

$$y = \begin{cases} x, & x > 0 \\ \alpha x, & \text{其他} \end{cases}$$

该函数等价于：

$$f(x) = \max(x, \alpha x)$$

从上面描述的 ReLU 相关激活函数来看，ReLU 是一种特殊的 Maxout 函数。

2.2.6 Maxout

Maxout 激活函数^[4]就是通常的最大值函数 \max ，即多个输入取最大值：

$$y = \max_k a_k = \max(w_1^T x + b_1, w_2^T x + b_2, \dots, w_n^T x + b_n)$$

其求导非常简单，取最大值的一路有梯度，其他路无梯度：

$$\frac{\partial y}{\partial a_i} = \begin{cases} 1, & a_i \text{ 为最大值} \\ 0, & \text{其他} \end{cases}$$

Maxout 能够在一定程度上缓解梯度消失的问题，同时又规避了 ReLU “死亡” 单元的问题，但是增加了参数和计算量。

2.2.7 Softmax

Softmax 通常应用在多分类问题的输出层，它可以保证所有输出神经元之和为 1，而每个输出对应的 $[0, 1]$ 区间的数值就是该输出的概率，在应用时取概率最大的输出作为最终的预测。Softmax 函数的形式如下：

$$y_j = \frac{e^{z_j}}{\sum_{t=1}^k e^{z_t}}$$

偏导数分为两种情况：

(1) $i = j$ 时

$$\frac{\partial y_i}{\partial z_i} = \frac{e^{z_i}(\sum_{t=1}^k e^{z_t}) - e^{z_i}e^{z_i}}{(\sum_{t=1}^k e^{z_t})^2} = y_i - y_i^2$$

(2) $i \neq j$ 时

$$\frac{\partial y_j}{\partial z_i} = \frac{0 - e^{z_i}e^{z_j}}{(\sum_{t=1}^k e^{z_t})^2} = 1 - y_i y_j$$

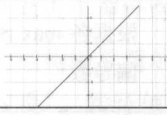

2.2.8 小结

激活函数是深度学习中非常重要的组成部分,也涌现出了非常多的研究成果,其中 ReLU 和 Maxout 的变种就有很多,以下简单罗列一些代表性工作。

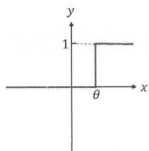




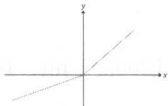
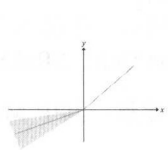
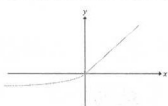

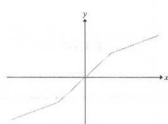
- 将 PReLU 中的 α 作为一定范围内随机变量的随机 Leaky ReLU (Randomized Leaky Rectified Linear Unit, RReLU) [5];
- 将横坐标左侧变成指数函数的指数线性单元 (Exponential Linear Unit, ELU) [6];
- 自适应的分段线性函数 APL (Adaptive Piecewise Linear) [7];
- 分段线性的 S 型 ReLU (S-Shaped Rectified Linear Activation Unit, SReLU) [8]。

表 2-2 总结了常见的相关激活函数图像以及对应的公式和偏导数。其中 Maxout 和 Softmax 涉及多路,省略了函数图像。

表 2-2 激活函数汇总^[9]

名称	图像	公式	导数
Identity		$f(x) = x$	$f'(x) = 1$
Linear		$f(x) = wx + b$	$f'(x) = w$

续表

名称	图像	公式	导数
Linear Threshold Neuron		$f(x) = \begin{cases} 1, & x \geq \theta \text{ 且无抑制突触输入} \\ 0, & \text{其他} \end{cases}$	$f'(x) = \begin{cases} ?, & x = \theta \\ 0, & \text{其他} \end{cases}$
Sigmoid		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$
ReLU		$f(x) = \begin{cases} x, & x > 0 \\ 0, & \text{其他} \end{cases}$	$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & \text{其他} \end{cases}$
Leaky ReLU		$f(x) = \begin{cases} x, & x > 0 \\ 0.01x, & \text{其他} \end{cases}$	$f'(x) = \begin{cases} 1, & x > 0 \\ 0.01, & \text{其他} \end{cases}$
PReLU		$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & \text{其他} \end{cases}$	$f'(x) = \begin{cases} 1, & x > 0 \\ \alpha, & \text{其他} \end{cases}$
RReLU		$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & \text{其他} \end{cases}$ $\alpha \sim U(l, u), l < u, l, u \in [0, 1)$ $U \text{ 为均匀分布}$	$f'(x) = \begin{cases} 1, & x > 0 \\ \alpha, & \text{其他} \end{cases}$
ELU		$f(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & \text{其他} \end{cases}$	$f'(x) = \begin{cases} 1, & x > 0 \\ \alpha e^x, & \text{其他} \end{cases}$
APL		$f(x) = \max(0, x) + \sum_{s=1}^S a_l^s \max(0, -x + b_l^s)$	根据各 max 函数的取值求偏导数
SReLU		$f(x) = \begin{cases} t_l + \alpha_l(x - t_l), & x \leq t_l \\ x, & t_l < x < t_r \\ t_r + \alpha_r(x - t_r), & x \geq t_r \end{cases}$	$f'(x) = \begin{cases} \alpha_l, & x \leq t_l \\ 1, & t_l < x < t_r \\ \alpha_r, & x \geq t_r \end{cases}$

续表

名称	图像	公式	导数
Maxout	—	$f(\mathbf{x}) = \max_k a_k$	$\frac{\partial f(\mathbf{x})}{\partial a_i} = \begin{cases} 1, & a_i \text{ 为最大值} \\ 0, & \text{其他} \end{cases}$
Softmax	—	$f(\mathbf{x})_j = \frac{e^{z_j}}{\sum_{t=1}^k e^{z_t}}$	$\frac{\partial f(\mathbf{x})_j}{\partial z_i} = f(\mathbf{x})_i(\delta_{ij} - f(\mathbf{x})_j)$ 其中: $\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & \text{其他} \end{cases}$

激活函数如此之多，一般在实践中选用哪些呢？针对这个问题网上有很多相关的讨论，但是结论并不统一。一般来说，推荐不用 Sigmoid 的较多，ReLU 要注意检查“死亡”单元的比例，PReLU 和 Maxout 等都是可以尝试的。此外，虽然理论上可以多种激活函数混用，但在实践中较少这样应用。

2.3 感知机

在了解了神经元的基本概念后，我们可以从计算机神经网络长长的源头——感知机开始，详细地了解这门学科及其跌宕起伏的历史。

2.3.1 感知机的提出

一切都要从 20 世纪 60 年代说起，Frank Rosenblatt 出版的《神经动力学原理：感知机和大脑机制的理论》揭开了人工神经网络研究的序幕。书中介绍了多种感知神经元，以及一套简洁却显得十分有效的学习算法。这本书引起了学术界和民众的极大兴趣，在那个科技腾飞的年代，人们对人工神经网络怀有极大的信心和期望。

感知机的结构极其简单。输入层为人们选择的“特征值”（feature），感知机内含一套参数，称为“权重”，特征值和权重相乘后求和，与一个阈值比较后输出为 0 或 1。回顾上一节所提到的神经元，相信读者对它已经不陌生了。以 $X = [x_1, x_2, \dots, x_n]$ 来表示输入值，以 $W = [w_1, w_2, \dots, w_n]$ 来表示对应参数，以 b 来表示阈值的相反数，以 y 来表示预测值，以 \bar{y} 来表示真实值，则感知机模型如图 2-4 所示。

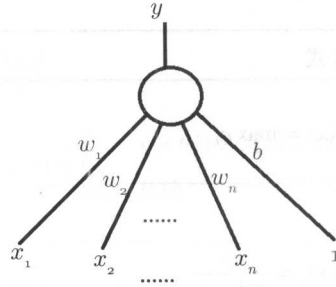


图 2-4 感知机模型

感知机模型的学习算法（即更新规则）也十分简明，如算法 2-1 所示。

算法 2-1 感知机模型的学习算法

随机取一个训练样本 (X, \tilde{y}) ，通过当前感知机模型进行预测。

- 如果预测结果正确，则当前感知机维持不变。
- 如果预测结果错误，则按下述规则更新感知机参数。
 - 如果样本真实输出值 $\tilde{y} = 1$ ，而实际预测值 $y = 0$ ，则将参数与该样本输入的“和”作为新的参数 $W \leftarrow W + X$ 。
 - 如果样本真实输出值 $\tilde{y} = 0$ ，而实际预测值 $y = 1$ ，则将参数与该样本输入的“差”作为新的参数 $W \leftarrow W - X$ 。

这套学习算法简单明了，并且保证模型不断优化收敛，直至满足所有的训练样本。直到后来人们才发现，这个论断过于草率。然而，在当时，一切还笼罩在虚假的繁荣之中。例如，当时专家宣称已经可以用计算机区分出“坦克”和“卡车”。这在当时是爆炸性的科技进展，但后来才发现，之所以能区分，是因为“坦克”的图片都是在晴天拍摄的，而“卡车”的图片都是在阴天拍摄的，感知机仅仅是计算了图片的平均光强，从而区分出这两类图片，并不是像人们期望的那样，真的对图片中的物体有了感知。这类事例给人工神经网络带来了非常负面的影响。

2.3.2 感知机的困境

1969 年，Minsky 和 Papert 对当时的感知机模型进行了深入的研究，并且出版了一本书，名字就叫《感知机》。书中分析了感知机“能做”和“不能做”的事情，这是对当时感知机理论很好的分析和总结。然而，人们普遍关注的是感知机“不能”完成的方面，并将该论述

扩大化,认为其是神经网络的普适困境,因而悲观地论断神经网络没有出路和研究价值。学术界对神经网络的研究开始冷淡。

具体来说,一方面,此类感知机模型仅能处理线性可分的问题,如果面对的问题并不是线性可分的,那么这种单层感知机是没有可行解的。比如,对于简单的“异或”问题,由于它是非线性问题,因而感知机是无法得到可行解的。读者可参考图 2-5 试试,是不是找不到一组可行解使得模型输出可以满足异或逻辑。另一方面,感知机需要人工提取特征作为输入。换个思路来看,对于非线性问题,感知机只有通过人工提取特定的特征——在这些特征中将非线性的因素包含进来——使得特征仅用线性关系就可判别,才能达到目标。但这意味着什么呢?这意味着很多问题的难点被转移到“特征的设定”上,而这一点又只能通过人工来完成,感知机完全帮不上忙。这个事实引起了人们对感知机实际功用的质疑。对于感知机的研究者来说,如何自动提取这类复杂的特征是摆在面前最迫切的问题。然而,从研究者开始想到解决方法的时候,距离陷入感知机困境的当年,已经过了长达 20 年的时间。

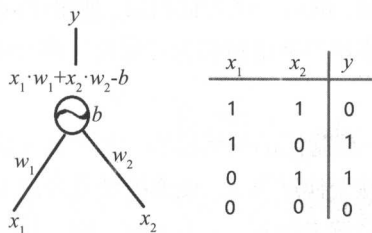


图 2-5 异或问题

2.4 DNN

解决感知机面临非线性问题的方法,就是将感知机变成多层神经网络,也称为深度神经网络(Deep Neural Network, DNN)。1974 年,Werbos 的博士论文^[10]证明了将感知机叠加起来组成神经网络,并且利用“后向传播”的方法,就可以解决诸如“异或问题”的非线性问题。如图 2-6 所示是其一可行的网络,读者可以验证看看。然而,该论文并没有引起广泛的关注。因为对于大多数研究者来说,多层神经网络相对于当时风头正劲的支持向量机(SVM)而言,其背后缺乏优美的数学理论和解决问题的坚实证明。实际上,这是神经网络一直面临的窘境。即使在其重获关注,并在各个领域获得划时代的成绩后,对于其“成功”的解释依然是一层未揭开的面纱。不过,随着近年来对神经网络的隐层的研究,如“可视化分析”等,人们正逐渐了解它背后神秘的机制。当然,这是后话。现在,就让我们走近神经网络基本模型和基本学习方法吧!

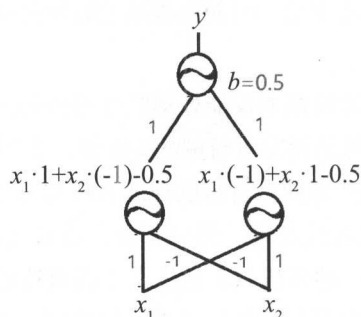


图 2-6 可以解决异或问题的一个多层神经网络

2.4.1 输入层、输出层及隐层

网络结构的第一层为输入层，最后一层为输出层，如果中间有其他层，则被称为“隐层”。如果隐层的数目多于一层，则该神经网络被称为“深度”神经网络。隐层和输出层一般会含有神经元，从而实现非线性。

如图 2-7 所示为一个带有一层隐层的网络模型。如果放大隐层或输出层的神经元，则可以将其分解为输入和参数的线性变换结果 z ，该变换结果经过非线性的激活函数 $y = f(z)$ 而得到输出的两部分结构。不过请读者注意， y 与 z 属于同一层，只是为了便于介绍神经网络的训练学习，我们才采取这种分解的结构来表示网络。

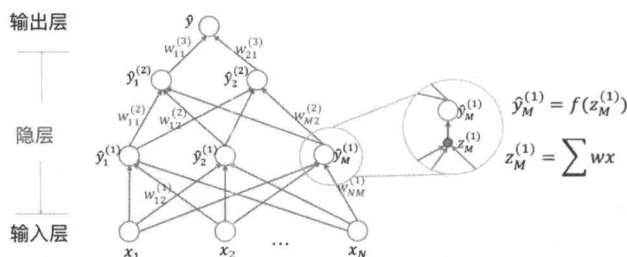


图 2-7 网络模型示例及放大的神经元

2.4.2 目标函数的选取

在讨论神经网络的训练之前，我们首先要明确目标函数。通常，这个目标函数以损失函数（Loss Function）的形式来呈现。例如，常用的均方误差损失函数可以表示为：

$$\text{Loss} = \frac{1}{2N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

其中, N 为样本的数目, y_i 为第 i 个样本的实际标注值, 也称为标签 (Label), 而 \hat{y}_i 为该样本的预测值。由此可见, 损失函数值越小越好, 当损失函数值为 0 时, 则说明模型预测的结果完全无误。

除均方差损失函数外, 还有很多不同的损失函数。损失函数的选取一般要根据模型的特点和目标的设立来进行。比如, 在一个多分类问题上, 最合适的损失函数可能就不是均方差损失函数。下面我们来具体看一下这个问题。

假设这个多分类问题共有 C 个类别, 而输出 z_c 也是 C 维的, 每一维的输出 z_c 值代表在该类的得分, 得分最高的即为最可能的预测类别。对于这样的问题, 我们更希望输出是概率形式, 因此, 在输出层会加一个 Softmax:

$$\hat{y}_c = \frac{\exp(z_c)}{\sum_i \exp(z_i)}$$

这样输出的预测值被转化成了概率值, 所有类的概率值之和为 1。对于这样的以 Softmax 为输出层的网络模型, 最合适的损失函数就是一种名为交叉熵的损失函数 (Cross-entropy Loss Function):

$$\text{Loss} = - \sum_{i=1}^N y_i \log(\hat{y}_i)$$

这是因为 $\frac{\partial \text{Loss}}{\partial z_i} = \sum_i \frac{\partial \text{Loss}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} = \hat{y}_i - y_i$ 。这个偏导数的结果简洁、漂亮。在网络的训练中很重要的就是利用 Loss 对参数的梯度, 而此简洁的梯度也让训练更加简单, 因此对于输出层为 Softmax 来说, 最合适的损失函数为此交叉熵损失函数。对于这个梯度公式的计算过程如下, 感兴趣的读者可以自己演算一遍。

$$\begin{aligned} \frac{\partial \text{Loss}}{\partial z_i} &= \sum_j \frac{\partial \text{Loss}}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i} \\ &= \sum_{j \neq i} \frac{\partial \text{Loss}}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_i} + \frac{\partial \text{Loss}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_i} \\ &= \sum_{j \neq i} -\frac{y_j}{\hat{y}_j} \cdot \frac{-\exp(z_j) \cdot \exp(z_i)}{(\sum_k \exp(z_k))^2} + \left(-\frac{y_i}{\hat{y}_i} \cdot \frac{\exp(z_i) \cdot (\sum_k \exp(z_k)) - \exp(z_i) \cdot \exp(z_i)}{(\sum_k \exp(z_k))^2} \right) \end{aligned}$$

$$\begin{aligned}
&= \sum_{j \neq i} \frac{y_j}{\hat{y}_j} \cdot \hat{y}_j \cdot \hat{y}_i + \left(-\frac{y_i}{\hat{y}_i} \right) \hat{y}_i (1 - \hat{y}_i) \\
&= \sum_{j \neq i} y_j \cdot \hat{y}_i + y_i \cdot \hat{y}_i - y_i \\
&= \hat{y}_i - y_i
\end{aligned}$$

有了目标损失函数以后，我们可以详细地看一下神经网络的训练过程——前向传播（Forward Propagation）与后向传播（Backward Propagation）。

2.4.3 前向传播

为了便于说明，我们以一个简单的神经网络模型为例。如图 2-8 所示为要训练的网络模型，其中输入为 N 维，输出为预测值 \hat{y} ，目标损失函数采用均方差误差，模型的参数为各层的 w 值，神经元的中间结果用 z 来表示，激活函数采用 Sigmoid。

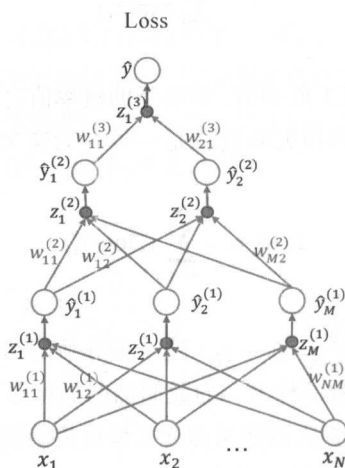


图 2-8 神经网络模型示例

前向传播，就是在当前参数值下，输入值进入网络后，顺序计算，最终得到预测值的过程。在图 2-8 中，则是自下向上沿着箭头方向进行计算的。其中：

$$\begin{cases} z_i^{(1)} = \sum_{j=1}^N w_{ji}^{(1)} x_j \\ \hat{y}_i^{(1)} = \frac{1}{1 + e^{-z_i^{(1)}}} \end{cases}$$

为输入层到第一隐层的前向传播计算公式。而第一隐层至第二隐层、第二隐层至输出层的前向传播计算也可以类似推得。最终的损失函数计算为输出值和真实值之间的均方误差或交叉熵误差等。

可以看出，前向传播计算非常简单。但是，所得到的预测值可能和真实值相差较远，其损失函数值也比较大。那么如何训练模型的参数使得损失值下降呢？这个重要的课题在神经网络中是以后向传播来实现的。

2.4.4 后向传播

后向传播，顾名思义，就是从损失值开始，反过来更新网络的参数值，使得更新后的网络的损失值下降的过程。这一过程主要是通过“梯度下降”的方法来实现的。也就是说，基于当前的参数值，能使损失值下降最大的方法可能是向着梯度的反方向更新参数。朝着这个梯度的反方向更新多大的值？更新是否一定能保证损失值减小？这些问题将在下一节中分析。本节我们先研究采取“梯度下降”策略后，如何得到梯度值，从而顺利完成后向传播的过程。

还是以图 2-8 所示的神经网络模型为例。我们先看相邻层之间的梯度计算，然后再看从损失值开始至任意层任意一个参数的梯度的计算方法。

首先，损失函数对输出层的梯度可以很容易求得：

$$\frac{\partial \text{Loss}}{\partial \hat{y}} = -(y - \hat{y})$$

那么，输出层 \hat{y} 对激活函数的输入 $z_1^{(3)}$ 的梯度是什么呢？因为我们选择的激活函数是 Sigmoid，而前面也已推得 Sigmoid 的导数结果，所以现在可以直接得到：

$$\frac{\partial \hat{y}}{\partial z_1^{(3)}} = \hat{y} \cdot (1 - \hat{y})$$

而由于 $z_j^{(3)}$ 是其对应的输入层在相应参数下的线性组合，因此其对参数和输入的偏导都很简单：

$$\begin{cases} \frac{\partial z_j^{(3)}}{\partial \hat{y}_i^{(2)}} = w_{ij}^{(3)} \\ \frac{\partial z_j^{(3)}}{\partial w_{ij}^{(3)}} = \hat{y}_i^{(2)} \end{cases}$$

而从 $y_i^{(2)}$ 层向下的梯度计算都是类似的：

$$\frac{\partial \hat{y}_i^{(2)}}{\partial z_i^{(2)}} = \hat{y}_i^{(2)} \cdot (1 - \hat{y}_i^{(2)})$$

从 $z_j^{(2)}$ 向下求梯度为：

$$\begin{cases} \frac{\partial z_j^{(2)}}{\partial \hat{y}_i^{(1)}} = w_{ij}^{(2)} \\ \frac{\partial z_j^{(2)}}{\partial w_{ij}^{(2)}} = \hat{y}_i^{(1)} \end{cases}$$

直到隐层向输入层的梯度计算：

$$\frac{\partial \hat{y}_i^{(1)}}{\partial z_i^{(1)}} = \hat{y}_i^{(1)} \cdot (1 - \hat{y}_i^{(1)})$$

一般最后只要求对参数的梯度（注：生成模型有时也需要对输入的梯度），而不再需要计算对输入值的梯度：

$$\frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}} = x_i$$

至此，我们得到了相邻两层梯度的计算结果。

而梯度下降法需要的是任意一个参数相对于损失值的梯度，关于这个梯度的计算，只需要应用梯度的“链式法则”，根据上面求得的结果便可以得到。比如，如果想求得损失值对参数 $w_{M2}^{(2)}$ 的梯度，则只需计算：

$$\begin{aligned} \frac{\partial \text{Loss}}{\partial w_{M2}^{(2)}} &= \frac{\partial \text{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial y_2^{(2)}} \cdot \frac{\partial y_2^{(2)}}{\partial z_2^{(2)}} \cdot \frac{\partial z_2^{(2)}}{\partial w_{M2}^{(2)}} \\ &= -(y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}) \cdot w_{21}^{(3)} \cdot \hat{y}_2^{(2)} \cdot (1 - \hat{y}_2^{(2)}) \cdot y_M^{(1)} \end{aligned}$$

而如果从损失值到变量有多条路径，那么就需要将各条路径上的梯度求出来，然后再加和。比如，如果要求损失值对 $\hat{y}_2^{(1)}$ 的梯度，从前向传播来看， $\hat{y}_2^{(1)}$ 的改变将会沿着两条路径影响到损失值，因此损失值对它的梯度计算应该是：

$$\frac{\partial \text{Loss}}{\partial \hat{y}_2^{(1)}} = \frac{\partial \text{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial \hat{y}_1^{(2)}} \cdot \frac{\partial \hat{y}_1^{(2)}}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial \hat{y}_2^{(1)}} + \frac{\partial \text{Loss}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1^{(3)}} \cdot \frac{\partial z_1^{(3)}}{\partial \hat{y}_2^{(2)}} \cdot \frac{\partial \hat{y}_2^{(2)}}{\partial z_2^{(2)}} \cdot \frac{\partial z_2^{(2)}}{\partial \hat{y}_2^{(1)}}$$

具体结果不再展开。

至此，后向传播的过程和梯度的具体计算我们已经清楚了。

2.4.5 参数更新

最后，我们再来看一下参数更新。虽然知道了参数更新的方向，并且由后向传播计算出了梯度值，但是沿着这个梯度的反方向更新多少还是一个重要的问题。在实际神经网络的训练中，往往会通过一个重要的参数——学习率（Learning Rate）来控制这个“步长”。也就是说，参数 w 的更新将通过以下表达式：

$$w \leftarrow w - \eta \cdot \frac{\partial \text{Loss}}{\partial w}$$

其中，“负号”代表与梯度方向相反； η 代表学习率，它作为参数来控制步长；而 $\frac{\partial \text{Loss}}{\partial w}$ 为计算的梯度值。

如图 2-9 所示，当前参数值为 x_t ，则下一时刻更新为 $x_{t+1} = x_t - \eta \cdot \partial y / \partial x$ 。具体来看，当前时刻该位置梯度的方向为 $+$ ，大小为 $\Delta y / \Delta x$ ，那么参数更新的方向将为 $-$ ，更新的大小为 $\eta \cdot \Delta y / \Delta x$ 。由图可见，如果学习率选取得当，更新后对应的 y 值将变小；但是当学习率过大时，也很容易出现 y 值反而增大的现象，发生震荡，如 x'_{t+1} 对应的值；而如果学习率设置得过小，那么虽然不易发生震荡，但收敛速度将会变慢，也会影响训练效果。由此可见，学习率的设置是一个十分重要的问题。在后面的章节中，我们会再详细探讨。

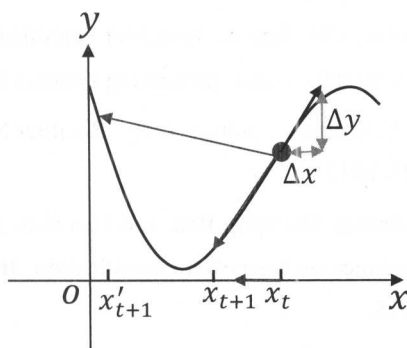


图 2-9 参数更新示例

2.4.6 神经网络的训练步骤

常用的基于梯度下降法的神经网络的训练步骤如算法 2-2 所示。

算法 2-2 神经网络的训练步骤

设计网络结构：输入层与输出层之间由几个隐层组成、各层之间如何连接，以及选择神经元。

选择网络参数的初始值及设立损失函数等。

开始训练：

1. 从训练集随机训练样本 X_i （如果是 batch 模式，则是多个训练样本）。
2. 将 X_i 在当前参数 W 下进行前向传播得到损失值（loss）。
3. 根据链式法则，进行后向传播得到梯度值 $\frac{\partial \text{loss}}{\partial w}$ 。
4. 更新参数值 $w \leftarrow w - \eta \cdot \frac{\partial \text{loss}}{\partial w}$ 。
5. 循环步骤 1~4，直至 loss 满足目标，网络训练完成。

至此，我们大体了解了神经网络的组成和基础训练步骤。后面将会有更深入、更有趣的内容等待我们研究。

参考文献

- [1] A Krizhevsky, I Sutskever, GE Hinton. Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems 2012, 1097-1105.
- [2] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. ICML 2013.
- [3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. IEEE International Conference on Computer Vision (ICCV), 2015.
- [4] IJ Goodfellow, D Wardefarley, M Mirza, A Courville, Y Bengio. Maxout Networks. Computer Science, 2013:1319-1327.
- [5] Xu B, Wang N, Chen T, et al. Empirical Evaluation of Rectified Activations in Convolutional Network[J]. Computer Science, 2015.

- [6] Clevert, Djork-Arné, Unterthiner T, Hochreiter S. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)[J]. Computer Science, 2015.
- [7] Agostinelli F, Hoffman M, Sadowski P, et al. Learning Activation Functions to Improve Deep Neural Networks[J]. Computer Science, 2014.
- [8] Jin X, Xu C, Feng J, et al. Deep Learning with S-shaped Rectified Linear Activation Units[J]. Computer Science, 2015, 3:1-8.
- [9] Activation Function. https://en.wikipedia.org/wiki/Activation_function.
- [10] Werbos, P. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974.

3

初始化模型

2006 年 Hinton 发表的 *Science* 论文^[1] 拉开了深度学习的序幕，这篇论文的主要贡献就是提出了一种深度模型的可行训练方法，其基本思想是利用生成模型受限玻尔兹曼机一层一层地进行初始化训练，然后再利用真实数据进行参数微调。

类似的训练方法使得之前广受质疑的深度模型不可训练的问题得到缓解，再加上后来的大数据（比如百万、千万级的 ImageNet 数据）以及高性能计算设备（比如 GPU），这些因素都为神经网络往更深层次发展奠定了坚实的基础。

本章只聚焦初始化模型，包括：受限玻尔兹曼机（Restricted Boltzmann Machine, RBM）、自动编码器（AutoEncoder, AE）、深度信念网络（Deep Belief Network, DBN）。

3.1 受限玻尔兹曼机

受限玻尔兹曼机由可视层和隐层构成，常常用来构建深层 RBM、自动编码器、深度信念网络等深度学习模型。RBM 属于深度学习中的生成模型，用于建模观察数据和输出标签之间的联合概率分布 $p(v, o)$ ，因而可以对 $p(o|v)$ 和 $p(v|o)$ 都进行评估，而判别模型仅对 $p(o|v)$ 进行估计。本章将要介绍的这几个生成模型主要流行于 2010 年左右，用于深度学习的初始化，而近两年来的生成模型——生成对抗网络（Generative Adversarial Network, GAN）大放异彩，使得大家又开始关注生成模型。关于 GAN 的介绍我们将留至本书的最后一章。

RBM 是一种概率图形式的神经网络模型，它是玻尔兹曼机（Boltzmann Machines, BM）的一种特殊形式，而玻尔兹曼机本身又是一种特殊的马尔可夫网络，马尔可夫网络又是一种

特殊的概率无向图模型，后者本身又是一种概率图模型。这些概念之间的关系如图 3-1 所示。RBM 也是一种特殊的能量模型，本章将从能量模型开始逐步引出 RBM 的具体解释。



图 3-1 受限玻尔兹曼机与其他概率图概念之间的关系

3.1.1 能量模型

基于能量的模型（Energy-Based Model, EBM），顾名思义，是描述一些感兴趣特征与能量之间关系的模型。一般来说，世间万物皆如此——系统越杂乱无序或者概率分布越趋近于均匀分布，系统对应的能量越大；系统越有序或者概率分布越集中，系统对应的能量越少；而当能量函数取最小值时，对应系统最稳定的状态。

基于能量的概率模型可以定义如下：

$$p(x) = \frac{e^{-E(x)}}{Z}$$

其中 Z 为归一化因子：

$$Z = \sum_x e^{-E(x)}$$

这个定义初看不好理解，但如果假设 $E(x) = -wx$ ，就不难看出此时的 EBM 其实就是 Softmax 模型，可以说 Softmax 是能量模型的一种特殊形式。

与逻辑回归（Logistic Regression）和 Softmax 类似，EBM 对应的 Log 似然如下：

$$\mathcal{L}(\theta, \mathcal{D}) = \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log p(x^{(i)})$$

其中 θ 为参数， \mathcal{D} 是大小为 N 的数据集， $x^{(i)}$ 为数据集中的第 i 个样本，一般 EBM 的损失函数定义为负的 Log 似然：

$$l(\theta, \mathcal{D}) = -\mathcal{L}(\theta, \mathcal{D}) = -\frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log p(x^{(i)})$$

对单个样本 x_i 来说，对应的 loss 为：

$$-\log p(x^{(i)})$$

那么模型中参数 θ 对应的梯度为：

$$\varepsilon_{\theta} = -\frac{\partial \log p(x^{(i)})}{\partial \theta}$$

3.1.2 带隐藏单元的能量模型

在很多情况下，并不能直接观测到所有的 x 值，这时候往往需要引入隐藏变量。假设给定输入 x 以及对应的隐藏变量 h ， $p(x)$ 可以重写为：

$$p(x) = \sum_h p(x, h) = \sum_h \frac{e^{-E(x, h)}}{Z}$$

假设 \mathcal{F} 为自由能量，定义如下：

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x, h)}$$

那么 $p(x)$ 则可改写为：

$$p(x) = \frac{e^{-\mathcal{F}(x)}}{Z}$$

其中归一化项 Z 为:

$$Z = \sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})}$$

单个样本 x 的梯度为:

$$\begin{aligned} g_{\theta} &= -\frac{\partial \log p(x)}{\partial \theta} = -\frac{\partial(-\mathcal{F}(x) - \log Z)}{\partial \theta} \\ &= \frac{\partial \mathcal{F}(x)}{\partial \theta} + \frac{1}{Z} \frac{\partial Z}{\partial \theta} \\ &= \frac{\partial \mathcal{F}(x)}{\partial \theta} + \frac{1}{Z} \frac{\partial(\sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})})}{\partial \theta} \\ &= \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{Z} \sum_{\tilde{x}} e^{-\mathcal{F}(\tilde{x})} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \\ &= \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \end{aligned}$$

第二项实际是计算 $\frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}$ 的期望:

$$E_p \left[\frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \right] = \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}$$

这里的思想和负采样 (Negative Sampling) 非常类似, 所看到的训练数据 x 是正样本, 根据概率分布 p 采样到的样本 \tilde{x} 是负样本。采样的目标是要 \tilde{x} 服从 p 分布, 实际就是在做蒙特卡洛 (Monte-Carlo) 采样。这些采样到的负例集合记作 \mathcal{N} , 梯度可以改写为如下形式:

$$g_{\theta} = -\frac{\partial \log p(x)}{\partial \theta} \approx \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{|\mathcal{N}|} \sum_{\tilde{x} \in \mathcal{N}} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta}$$

有了上面的公式, 我们就可以借助 MCMC (Markov Chain Monte Carlo) 之类的采样方法学习得到一个 EBM 模型。

3.1.3 受限玻尔兹曼机基本原理

玻尔兹曼机 (Boltzmann Machine, BM) 是一种特殊的对数线性马尔可夫随机场 (Log-linear Markov Random Field), 因为其能量函数是参数的线性形式。它包含 m 个观察单元 $\mathbf{v} = (v_1, \dots, v_m)$ 及 n 个隐藏单元 $\mathbf{h} = (h_1, \dots, h_n)$, 其中隐藏单元既要依赖于观察单元, 也要依赖于其他隐藏单元; 观察单元可能既依赖于隐藏单元, 也依赖于同层的其他观察单元。

受限玻尔兹曼机（Restricted Boltzmann Machine, RBM）是一种特殊的玻尔兹曼机，即同层之间不存在相互依赖关系，如图 3-2 所示，只有观察层和隐藏层之间存在关联。

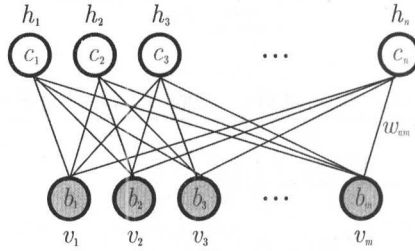


图 3-2 受限玻尔兹曼机

RBM 对应的能量函数 $E(\mathbf{v}, \mathbf{h})$ 为：

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{h}^T \mathbf{W} \mathbf{v} - \mathbf{b}^T \mathbf{v} - \mathbf{c}^T \mathbf{h} = -\sum_{i=1}^n \sum_{j=1}^m w_{ij} h_i v_j - \sum_{j=1}^m b_j v_j - \sum_{i=1}^n c_i h_i$$

其中 w_{ij} 表示第 i 个隐藏单元与第 j 个观察单元之间的连接权重， b_j 和 c_i 则是对应的 bias（偏置）项。

自由能量 $\mathcal{F}(\mathbf{v})$ 计算如下：

$$\begin{aligned} \mathcal{F}(\mathbf{v}) &= -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} = -\log \sum_{\mathbf{h}} e^{\mathbf{b}^T \mathbf{v} + \mathbf{c}^T \mathbf{h} + \mathbf{h}^T \mathbf{W} \mathbf{v}} \\ &= -\mathbf{b}^T \mathbf{v} - \sum_i \log \sum_{h_i} e^{h_i (c_i + W_i \mathbf{v})} \end{aligned}$$

从概率图的角度来看，给定所有观察变量的值时隐藏变量之间相互独立；对称的，给定所有隐藏变量的值时观察变量之间相互独立，即：

$$\begin{aligned} p(\mathbf{h}|\mathbf{v}) &= \prod_{i=1}^n p(h_i|\mathbf{v}) \\ p(\mathbf{v}|\mathbf{h}) &= \prod_{i=1}^m p(v_i|\mathbf{h}) \end{aligned}$$

下面讨论在 RBM 中如何对参数求梯度：

$$p(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}$$

沿用能量模型的相关结论，损失函数为：

$$\begin{aligned} l(\theta|\mathbf{v}) &= -\mathcal{L}(\mathbf{v}|\theta) = -\log p(\mathbf{v}|\theta) = -\log \left(\frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) = -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} + \log Z \\ &= -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} + \log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \end{aligned}$$

对参数的偏导为：

$$\begin{aligned} \frac{\partial l(\theta|\mathbf{v})}{\partial \theta} &= -\frac{\partial (\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})})}{\partial \theta} + \frac{\partial (\log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})})}{\partial \theta} \\ &= \frac{1}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} - \frac{1}{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \\ &= -\sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} + \sum_{\mathbf{v}, \mathbf{h}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \end{aligned}$$

3.1.4 二值 RBM

如果规定所有观察变量 v_i 和隐藏变量 h_j 的取值范围都为 $\{0, 1\}$ ，那么这样得到的 RBM 一般称为二值 RBM，有时也称为伯努利-伯努利 RBM。

关于二值 RBM，我们可以得到以下公式：

$$\begin{aligned} \therefore p(\mathbf{v}, \mathbf{h}) &= \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z} \\ \therefore p(\mathbf{v}, h_i = 1) &= \frac{e^{-E(\mathbf{v}, 1)}}{Z} = \frac{e^{\mathbf{b}^T \mathbf{v} + c_i + W_i^T \mathbf{v}}}{Z} \\ p(\mathbf{v}, h_i = 0) &= \frac{e^{-E(\mathbf{v}, 0)}}{Z} = \frac{e^{\mathbf{b}^T \mathbf{v}}}{Z} \\ p(h_i = 1|\mathbf{v}) &= \frac{p(\mathbf{v}, h_i = 1)}{p(\mathbf{v}, h_i = 1) + p(\mathbf{v}, h_i = 0)} \\ &= \frac{e^{\mathbf{b}^T \mathbf{v} + c_i + W_i^T \mathbf{v}}}{e^{\mathbf{b}^T \mathbf{v} + c_i + W_i^T \mathbf{v}} + e^{\mathbf{b}^T \mathbf{v}}} \\ &= \frac{1}{1 + e^{-(c_i + W_i^T \mathbf{v})}} \\ &= \sigma(c_i + W_i^T \mathbf{v}) \end{aligned}$$

其中 σ 是 Sigmoid 函数, 同理:

$$p(v_j = 1 | \mathbf{h}) = \sigma(b_j + \mathbf{W}_j^T \mathbf{h})$$

自由能量 $\mathcal{F}(\mathbf{v})$ 对应计算如下:

$$\begin{aligned}\mathcal{F}(\mathbf{v}) &= -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} = -\log \sum_{\mathbf{h}} e^{\mathbf{b}^T \mathbf{v} + \mathbf{c}^T \mathbf{h} + \mathbf{h}^T \mathbf{W} \mathbf{v}} \\ &= -\mathbf{b}^T \mathbf{v} - \sum_i \log \sum_{h_i} e^{h_i(c_i + \mathbf{W}_i^T \mathbf{v})} \\ &= -\mathbf{b}^T \mathbf{v} - \sum_i \log(1 + e^{(c_i + \mathbf{W}_i^T \mathbf{v})})\end{aligned}$$

二值 RBM 的相关梯度计算如下。

前面已知:

$$\frac{\partial l(\theta | \mathbf{v})}{\partial \theta} = - \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} + \sum_{\mathbf{v}, \mathbf{h}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}$$

当 $\theta = w_{ij}$ 时, 第一项即为:

$$\begin{aligned}- \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial w_{ij}} &= - \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) \frac{\partial (\sum_{i=1}^n \sum_{j=1}^m w_{ij} h_i v_j - \sum_{j=1}^m b_j v_j - \sum_{i=1}^n c_i h_i)}{\partial w_{ij}} \\ &= - \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) v_j h_i\end{aligned}$$

也就是训练数据给定时 $v_j h_i$ 的期望, Hinton 在参考文献 [3] 中将此项记作 $\langle v_j h_i \rangle_{\text{data}}$, 表示的是在数据中观察变量 v_i 和隐藏变量 h_j 同时为 1 的频率。

同理, 梯度的第二项为:

$$\sum_{\mathbf{h}, \mathbf{v}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial w_{ij}} = \sum_{\mathbf{h}, \mathbf{v}} p(\mathbf{v}, \mathbf{h}) v_j h_i$$

也就是模型给定时 $v_j h_i$ 的期望, Hinton 在参考文献 [3] 中将此项记作 $\langle v_j h_i \rangle_{\text{model}}$, 表示的是模型预估的观察变量 v_i 和隐藏变量 h_j 同时为 1 的频率的期望值, 这个期望值是用最终模型定义的联合概率分布 $p(\mathbf{v}, \mathbf{h})$ 来计算的。

因此, 总的负梯度为:

$$-\frac{\partial l(\theta|\mathbf{v})}{\partial w_{ij}} = \langle v_j h_i \rangle_{\text{data}} - \langle v_j h_i \rangle_{\text{model}}$$

如果采用最简单的梯度下降方法, w_{ij} 的每次更新为:

$$\Delta w_{ij} = \eta (\langle v_j h_i \rangle_{\text{data}} - \langle v_j h_i \rangle_{\text{model}})$$

其中 η 是学习率 (Learning Rate)。

类似地, bias 变量的相关负梯度分别为:

$$\begin{aligned} -\frac{\partial l(\theta|\mathbf{v})}{\partial b_j} &= \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial b_j} - \sum_{\mathbf{v}, \mathbf{h}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial b_j} = v_j - \sum_{\mathbf{v}} p(\mathbf{v}) v_j \\ -\frac{\partial l(\theta|\mathbf{v})}{\partial c_i} &= \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial c_i} - \sum_{\mathbf{v}, \mathbf{h}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial c_i} = p(H_i = 1|\mathbf{v}) - \sum_{\mathbf{v}} p(\mathbf{v}) p(H_i = 1|\mathbf{v}) \end{aligned}$$

为了避免第二项中的期望, 可以借鉴采样的近似方法。

3.1.5 对比散度

由于隐层未知且 \mathbf{v}, \mathbf{h} 的组合空间很大, 联合概率分布 $p(\mathbf{v}, \mathbf{h})$ 是很难计算的, 因而 $\langle v_j h_i \rangle_{\text{model}}$ 往往采用采样 (如 MCMC 采样) 的方法。一般 MCMC 方法需要较多的采样步数, 而针对 RBM 训练的对比散度 (Contrastive Divergence, CD) [1][2][3] 算法只需要较少的步数, 其中 k 步对比散度近似为:

$$\langle v_j h_i \rangle_{\text{model}} = \langle v_j h_i \rangle_{\infty} \approx \langle v_j h_i \rangle_k$$

其中 $\langle \cdot \rangle_{\infty}$ 表示无穷步采样后得到的期望值, $\langle \cdot \rangle_k$ 表示 k 步采样后得到的期望值, 如图 3-3 所示。

k 步对比散度 (k-Step Contrastive Divergence, CD-k) 以训练样本 $\mathbf{v}^{(0)}$ 为起点, 执行 k 步吉布斯 (Gibbs) 采样, 其中第 t ($t = 0, 1, \dots, k-1$) 步执行如下操作:

- 利用 $p(\mathbf{h}|\mathbf{v}^{(t)})$ 采样出 $\mathbf{h}^{(t)}$ 。
- 利用 $p(\mathbf{v}|\mathbf{h}^{(t)})$ 采样出 $\mathbf{v}^{(t+1)}$ 。

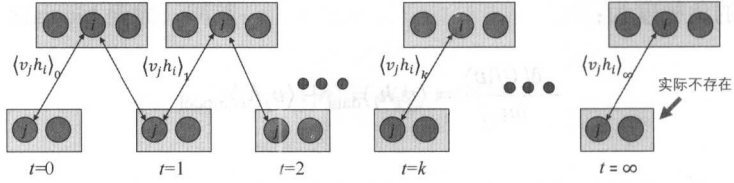


图 3-3 RBM 采样示例

然后利用第 $k-1$ 步采样得到的 $\mathbf{v}^{(k)}$ 来近似计算前面的梯度：

$$\begin{aligned} -\frac{\partial l(\theta|\mathbf{v})}{\partial w_{ij}} &= \langle v_j h_i \rangle_{\text{data}} - \langle v_j h_i \rangle_{\text{model}} \approx p(H_i = 1 | \mathbf{v}^{(0)}) v_j^{(0)} - p(H_i = 1 | \mathbf{v}^{(k)}) v_j^{(k)} \\ -\frac{\partial l(\theta|\mathbf{v})}{\partial b_j} &= v_j - \sum_v p(v) v_j \approx v_j^{(0)} - v_j^{(k)} \\ -\frac{\partial l(\theta|\mathbf{v})}{\partial c_i} &= p(H_i = 1 | \mathbf{v}) - \sum_v p(v) p(H_i = 1 | \mathbf{v}) \approx p(H_i = 1 | \mathbf{v}^{(0)}) - p(H_i = 1 | \mathbf{v}^{(k)}) \end{aligned}$$

具体的算法如算法 3-1 所示。第一个内部 for 循环是根据 h_i 和 v_j 的分布进行采样的；第二个内部 for 循环则根据公式计算梯度。

算法 3-1 k 步对比散度算法

输入：RBM($V_1, \dots, V_m, H_1, \dots, H_n$)，训练 Batch 数据集为 S

输出：变量的梯度近似为 Δw_{ij} 、 Δb_j 、 Δc_i ($i = 1, \dots, n, j = 1, \dots, m$)

初始化： $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \dots, n, j = 1, \dots, m$

foreach S 中的每条数据 \mathbf{v} :

$\mathbf{v}^{(0)} \leftarrow \mathbf{v}$

for $t = 0, \dots, k-1$ do

1. 利用 $p(\mathbf{h}|\mathbf{v}^{(t)})$ 采样出 $\mathbf{h}^{(t)}$
2. 利用 $p(\mathbf{v}|\mathbf{h}^{(t)})$ 采样出 $\mathbf{v}^{(t+1)}$

for $i = 1, \dots, n, j = 1, \dots, m$ do

1. $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 | \mathbf{v}^{(0)}) v_j^{(0)} - p(H_i = 1 | \mathbf{v}^{(k)}) v_j^{(k)}$
2. $\Delta b_j \leftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$
3. $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 | \mathbf{v}^{(0)}) - p(H_i = 1 | \mathbf{v}^{(k)})$

3.2 自动编码器

自动编码器 (AutoEncoder) 是一种无监督学习的算法。普通的监督学习方法如图 3-4 所示, 每个输入都有对应的标签, 即标注的期望值, 模型的预测值与期望值进行相关比较 (比如绝对值差或平方差等) 就可以得到对应的损失值 (loss), 而相关比较对应的函数就是机器学习中常说的损失函数。如图 3-5 所示则是一种无监督学习方法 (注: 并不是所有的无监督学习方法都是这样的, 比如聚类、PCA 降维等), 标签直接用输入代替。自动编码器一般由编码器 (Encoder) 网络和解码器 (Decoder) 网络两部分组成, 其中编码器网络在训练和线上部署时都被使用, 而解码器网络只在训练时被使用。如果将图 3-5 中的模型替换为与自动编码器相关的编码器和解码器, 就得到了如图 3-6 所示的自动编码器学习方法。

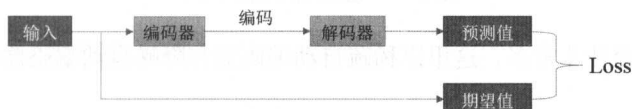


图 3-4 监督学习方法

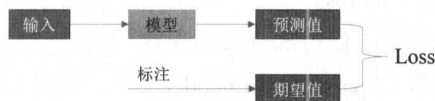


图 3-5 无监督学习方法

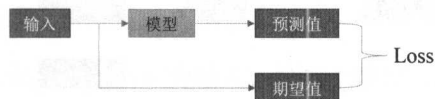


图 3-6 自动编码器学习方法

如图 3-7 所示是一个五输入的自动编码器, 输入为 $\{x_1, x_2, x_3, x_4, x_5\}$, 其中 $x_i \in \mathbf{R}$, 目标是学到函数 $h(\mathbf{x}) \approx \mathbf{x}$ 。自动编码器实质是以输入作为标签进行训练的, 从模型的结构来看, 输入向量 (在图 3-7 中维度为 5) 被编码为隐层向量 (在图 3-7 中维度为 3), 隐层向量又被解码为输出向量 (与输入向量维度相同)。这个例子中的每一层都是普通的全连接层, 当然, 自动编码器的层也可以是卷积层之类的, 如果是卷积层, 就是卷积自动编码器。

一般来说, 隐层向量的维度要小于输入向量的维度, 而因为隐层向量能基本还原出输入向量, 说明隐层向量保留了输入的绝大部分信息, 从而达到对输入进行降维的效果, 这有点

类似于传统机器学习中的主成分分析（PCA）等降维技术。

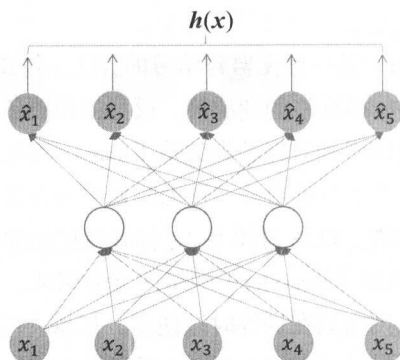


图 3-7 自动编码器示例

自动编码器的变种非常多，这里以稀疏自动编码器和降噪自动编码器为代表进行讲解。

3.2.1 稀疏自动编码器

如果对编码加入 L1 正则（即 loss 增加一项 $\frac{\lambda}{n} \sum_{i=1}^n \|\text{code}_i\|_1$ ，其中 $\|\text{code}_i\|_1$ 表示第 i 个样本对应编码的 L1 范数），就可以使得部分编码为 0，从而达到编码稀疏的效果，对应的自动编码器则为稀疏自动编码器（Sparse AutoEncoder）^[5]，如图 3-8 所示。

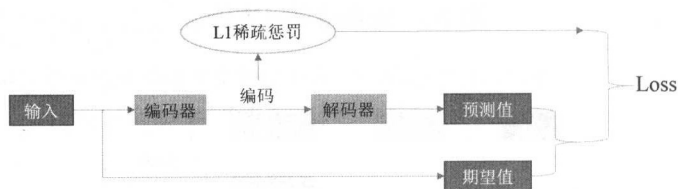


图 3-8 稀疏自动编码器学习方法

3.2.2 降噪自动编码器

降噪自动编码器（Denoising AutoEncoder）^[6]也是自动编码器的一个变种，唯一的区别在于编码器的输入是包含噪声的，而用作解码目标的输入是去除了噪声的，如图 3-9 所示。这样的编码器模型能够将有噪数据还原为干净的原始数据，从而具有较强的抗噪能力。

由于噪声数据和原始数据的标注成本往往是非常高的，所以通常的做法是通过对原始数据人为添加噪声参与训练，这些方法包括：

- 添加高斯噪声。
- 添加二维掩码噪声，类似于 Dropout，将部分输入神经元直接置为 0。

还有一些自动编码器的变种与降噪自动编码器的动机一样，都是增加学习的鲁棒性，比如通过修改损失函数的收缩自动编码器（Contractive AutoEncoder）。

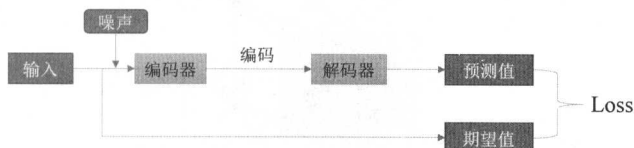


图 3-9 降噪自动编码器学习方法

3.2.3 栈式自动编码器

既然单层的编码能够尽量保留输入层的信息，如果在第一层编码的基础上继续构建一层自动编码器，那么新的编码能够尽量保留第一层编码的信息，也就能保留输入的绝大部分信息，这种叠加的自动编码器称为栈式自动编码器（Stacked AutoEncoder）。

利用栈式自动编码器进行逐层的贪婪训练方式非常适合深度学习模型的权重初始化。如图 3-10 所示是一个普通的四层 DNN 网络，可以采用如下栈式自动编码器的方式进行参数的初始化。

- 利用输入层和第一个隐层构建一个自动编码器，使得第一个隐层 $\langle h_{11}, h_{12}, h_{13}, h_{14} \rangle$ 被解码后的输出尽量与输入信息接近，如图 3-11 所示。如此得到的第一个隐层 $\langle h_{11}, h_{12}, h_{13}, h_{14} \rangle$ 充分保留了原始输入的信息。
- 以第一个隐层 $\langle h_{11}, h_{12}, h_{13}, h_{14} \rangle$ 为输入，第二个隐层 $\langle h_{21}, h_{22}, h_{23} \rangle$ 作为编码构建第二个自动编码器，类似地，最终学到的第二个隐层会最大程度地保留第一个隐层的信息，也就是在很大程度上保留了输入的信息。
- 依此类推，最终得到的第三个隐层 $\langle h_{31}, h_{32}, h_{33} \rangle$ 也会最大程度地保留输入信息，也就是得到了足够有效的特征。
- 将第三个隐层与 Softmax 层相组合，就可以训练一个简单的分类模型。
- 将以上所有层组合在一起就可以得到一个已经初始化的 DNN 模型。
- 有监督微调：使用了初始化参数的 DNN 模型可以进一步利用梯度下降之类的优化算法微调所有参数，从而使得模型达到更好的效果。

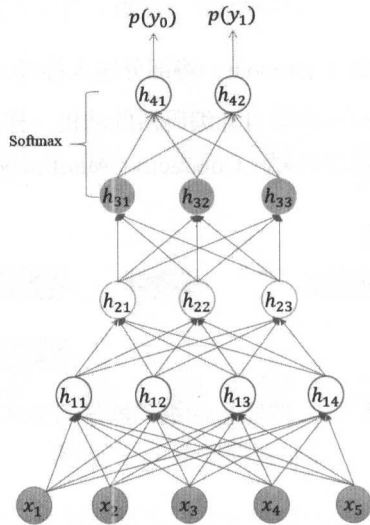


图 3-10 利用栈式自动编码器构建的四层 DNN 网络

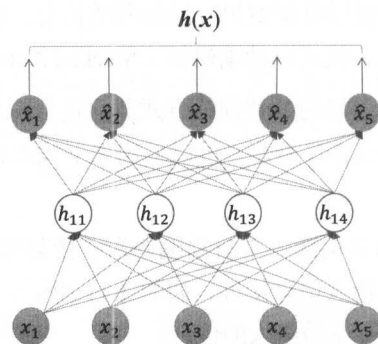


图 3-11 利用自动编码器初始化第一个隐层

3.3 深度信念网络

信念网络 (Belief Network) 又称为贝叶斯网络 (Bayesian Network) 或贝叶斯信念网络 (Bayesian Belief Network), 是一种有向无环图模型。这种模型可以在任意叶子节点生成无偏的样本集合, 因此认为这样的网络具有自己的“信念” (Belief)。深度信念网络 (Deep Belief Network, DBN) 是通过不断累积 RBM 形成的深层网络结构, 由 Geoffrey Hinton 等人于 2006 年首次提出^[7]。每当一个 RBM 被训练完成时, 其隐藏单元又可以作为后一层 RBM 的输入。DBN 的基本思想是允许每一层 RBM 模型接收数据的不同表示。

DBN 根据应用需求不同，既可以用作自动编码器，也可以用作分类器^[8]。

如图 3-12 所示为多层 RBM 逐层训练得到的 DBN，其中输入作为最底层，逐层进行 RBM 的无监督训练，下一层 RBM 的隐层输出作为上一层 RBM 的输入，当训练停止时，就拥有了 DBN 所有隐层权重的初始值，最后一个 RBM 的隐层输出（图 3-12 中的 f_1, f_2 ）就是最终的自动编码器的输出向量。

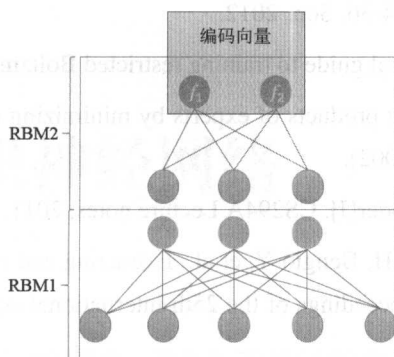


图 3-12 拥有两层 RBM 的自动编码器 DBN

DBN 也可以用作分类器，如图 3-13 所示，逐层预训练完之后，采用后向传播技术针对分类目标进行参数的微调。

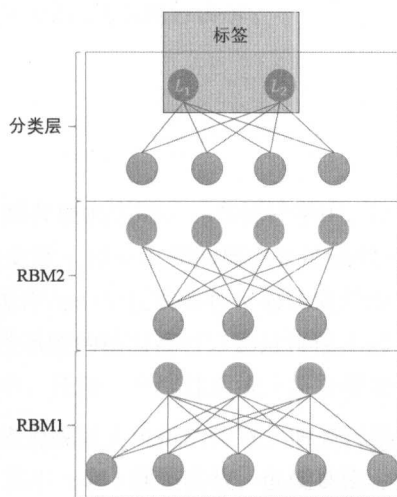


图 3-13 拥有两层 RBM 的分类器 DBN

参考文献

- [1] Hinton G E, Salakhutdinov R R. Reducing the dimensionality of data with neural networks[J]. science, 2006, 313(5786): 504-507.
- [2] A. Fischer, C. Igel. An introduction to restricted Boltzmann machines, Proc. 17th Iberoamer. Congr. Pattern Recognit., pp. 14-36, Sep. 2012.
- [3] G. E. Hinton, A practical guide to training restricted Boltzmann Machines, 2010.
- [4] Hinton, G.E.. Training products of experts by minimizing contrastive divergence. Neural Computation 14, 1771-1800 (2002).
- [5] Ng A. Sparse autoencoder[J]. CS294A Lecture notes, 2011, 72(2011): 1-19.
- [6] Vincent P, Larochelle H, Bengio Y, et al. Extracting and composing robust features with denoising autoencoders[C]. Proceedings of the 25th international conference on Machine learning. ACM, 2008: 1096-1103.
- [7] Hinton G E, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets[J]. Neural computation, 2006, 18(7): 1527-1554.
- [8] Keyvanrad M A, Homayounpour M M. A brief survey on deep belief networks and introducing a new object oriented toolbox (DeeBNet)[J]. arXiv preprint arXiv:1408.3264, 2014.

4

卷积神经网络

卷积神经网络并不是一个新的概念，甚至在 20 世纪 90 年代就已经被广泛应用，但深度学习卷土重来的第一功臣非卷积神经网络莫属，原因之一就是卷积神经网络是非常适合计算机视觉应用的模型。

这一章将带大家深入了解卷积神经网络的基本原理，包括卷积算子、卷积的特征、卷积神经网络的典型结构，以及其中的卷积层和池化层。而近年来涌现的经典卷积神经网络模型，我们将在计算机视觉部分的第 10 章进行讲解。

4.1 卷积算子

卷积^[1]在工程和数学上都有很多应用——在统计学中，加权的滑动平均是一种卷积；在概率论中，两个统计独立的变量 x 和 y 求和的概率密度函数是 x 和 y 的概率密度函数的卷积；在声学中，回声可以用原声与一个反映各种反射效应的函数相卷积来表示；在电子工程与信号处理中，任意一个线性系统的输出都可以通过将输入信号与系统函数（系统的应激响应）做卷积获得；在物理学中，任何一个线性系统（符合叠加原理）都存在卷积。

假设我们用激光传感器来追踪航天飞机的位置。激光传感器能够提供一个输出 $x(t)$ （表示 t 时刻航天飞机的位置），其中 x 和 t 都是实数，也就是说，我们可以在任意时刻从传感器中获得一个不同的读数。

现在，假设激光传感器受到一定的噪声影响。为了获取包含较少噪声的航天飞机位置的估计，我们期望对多个测量进行平均。当然，时刻越接近的测量越相关，因此期望以一种加

权平均的方式对较近的读数提供更大的权重。我们通过一个权重函数 $\omega(a)$ 来实现, a 表示测量值产生的时间间隔。如果将这样的加权平均操作应用在每个时刻上, 就得到了一个新的函数 s , 提供一种关于航天飞机位置的平滑估计:

$$s(t) = \int x(a) \omega(t-a) da$$

这种操作称作“卷积”。卷积运算通常用星号来标记:

$$s(t) = (x * \omega)(t)$$

其中, ω 必须是一种有效的概率密度函数才能使得函数输出为加权平均。并且, ω 在自变量为负的区间取值为 0, 否则将会使用未来的读数对当前值进行加权, 这种情况是超越了现实系统能力的。当然, 这些限制仅仅是针对上面的例子的。通常, 卷积被定义为对任意函数进行如上形式的积分, 并可能被应用于除加权平均之外的用途。

在卷积网络的术语中, 卷积的第一个参数(上例中的函数 x)通常表示输入(Input), 第二个参数(上例中的函数 ω)表示核(Kernel)。输出有时也被称作特征映射(Feature Map)。

通常, 用计算机处理数据时, 时间是离散的, 且传感器以固定的间隔提供读数。在航天飞机这个例子中, 要求激光传感器在每个时刻都提供测量结果并不现实, 比较现实的方案是传感器每秒提供一个测量结果且时间指数 t 只取正数值。如果假设 x 和 ω 只能定义在正数值 t 上, 我们就得到了离散卷积:

$$s(t) = (x * \omega)(t) = \sum_{a=-\infty}^{\infty} x(a) \omega(t-a)$$

在机器学习的应用中, 输入数据通常是一个多维数组, 核通常是一个通过学习算法获得的多维的参数数组。我们将这些多维数组称为张量(Tensor)。在实际中, 张量的维度都是有限的, 换句话说, 在实践中上式的无穷加和往往退化为有限数量的元素之和。

卷积也可以是多维的。例如, 如果将一个二维图像 I 作为输入, 也许同样希望使用一个二维的卷积核 K :

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i-m, j-n)$$

卷积运算具有交换性，这意味着可以等价地写作：

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n)$$

通常因为在 m 和 n 的有效值范围内变化更小，所以在机器学习库里会更直接地使用后面的公式形式。

卷积满足交换律是因为相对于输入其翻转了核函数，这样当 m 增加时，输入函数的下标增加，而核函数的下标则相应减小。对核进行翻转的唯一原因是为了获得交换性。尽管交换律在书写证明时很有用，但是在神经网络的应用中却不是重要的属性。

取而代之的是，在许多神经网络库的实现中采用一种称为互相关（Cross-correlation）的相关函数，它除不翻转核函数之外跟卷积一样：

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

许多机器学习库将互相关的实现称为卷积。本书中我们将两种操作都统称为卷积，并在与翻转有关的部分特别指明是否需要翻转核函数。在机器学习中，不论卷积核是否进行翻转，学习算法都能够学到恰当的参数。

如图 4-1 所示是针对二维输入进行卷积的示例（卷积核无翻转）。

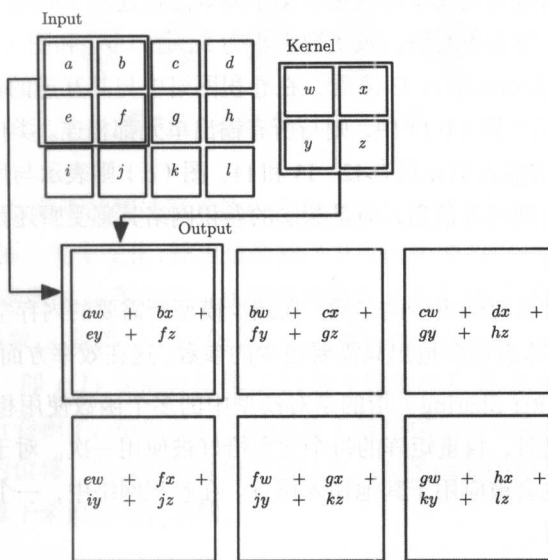


图 4-1 不带翻转的二维卷积

4.2 卷积的特征

卷积提供了能够提升机器学习效果的三个重要方法：稀疏交互（Sparse Interaction）或稀疏连接（Sparse Connectivity）、参数共享（Parameter Sharing）以及等价表达（Equivariant Representation）。此外，卷积也提供了一种使得输入尺寸可变的工作方式。本节将依次阐述这些观点。

传统的神经网络利用参数矩阵的乘法，每个输入单元和输出单元之间由单独的参数进行描述。这意味着输出单元会和每个输入单元进行交互，即连接是稠密的。而卷积网络因为核函数尺寸一般小于输入大小，其连接是稀疏的。例如，当处理一张图片时，输入图像可能包含上百万个像素，但是我们可以检测小的、有意义的特征，使用只占几十个或几百个像素的边缘卷积核。这种空间上的连接范围（作为神经元的超参数）称为“可视野（Receptive Field）”（与卷积核尺寸相同）。越高层的卷积层“可视野”对应到原始输入图像上的区域越大，也为提取到更高层的语义信息提供了可能。如果将神经网络和脑科学进行类比，这样的局部连接类似于人大脑视觉皮层不同位置只对局部区域有响应的生理机制。

下面举例对比全连接和稀疏连接的区别。假设有 m 路输入和 n 路输出，那么全连接的矩阵相乘需要 $m * n$ 个参数，算法复杂度为 $O(m * n)$ 。如果限制到每个输出单元的连接数为 k ，那么稀疏的连接方式只需要 $k * n$ 个参数和 $O(k * n)$ 的运行时间。对于许多现实应用，在保持 k 比 m 小几个数量级的情况下，机器学习任务仍然能够获得一个好的表现。这种稀疏连接特性使得网络能够通过构建简单的连接结构来高效地描述变量间的复杂关系。如图 4-2 所示，图（a）和图（c）为卷积网络，核函数大小为 3，图（b）和图（d）为普通的全连接神经网络。其中图（a）表示从输入 I3 来看，在卷积网络中与其互连的输出单元只有 O2、O3 和 O4；而在全连接网络（图（b））中，则与所有输出单元都相连。类似地，从输出 O3 来看，图（c）表示与其互连的输入单元只有 I2、I3 和 I4，图（d）则表示与所有输入单元相连。虽然卷积网络单层只能看到局部信息，但是深层的卷积网络其感受野还是可以涉及全图的，如图 4-3 所示。

这意味着我们只需要存储更少的参数，在减少模型所需要的内存空间的同时提高了模型的统计效率。这同样意味着计算输出只需要更少的参数。这在效率方面的提升通常十分显著。

参数共享（Parameter Sharing）指的是对模型中的多个函数使用相同的参数。在传统的神经网络中，计算输出时，权重矩阵的每个元素恰好被使用一次。对于共享参数，应用于一个输入位置的参数值也会被应用于其他输入位置。在卷积网络中，一个卷积核内的参数会被应用于输入的所有位置。

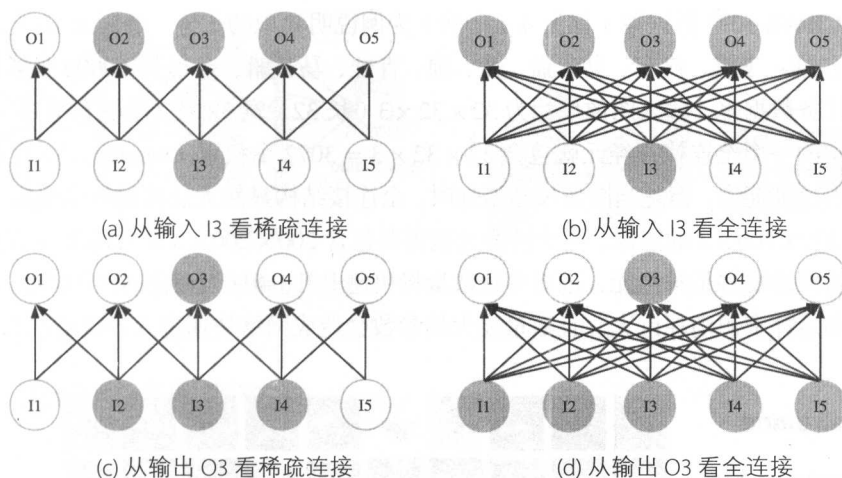


图 4-2 稀疏连接与全连接

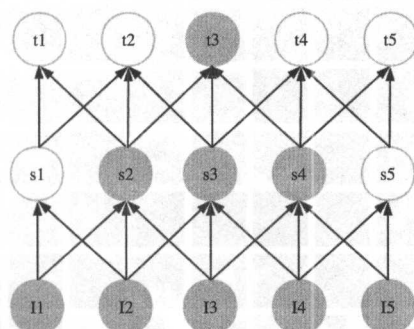


图 4-3 多层卷积的感受野

在卷积网络中，参数共享的特殊形式使得卷积层具有变换等价性 (Equivariance to Translation)。这意味着如果输入发生变化，输出也会随之发生同样的变化。特别地，如果 $f(g(x)) = g(f(x))$ ，则函数 f 与函数 g 是等价的。在卷积网络中，我们用 g 表示对输入的一种变换，那么卷积函数 f 与函数 g 是等价的。例如， I 表示一幅图像， $I' = g(I)$ 表示图像函数 $I'(x, y) = I(x - 1, y)$ ，即 $g(I)$ 变换将图像 I 的每个像素向右移动一个单元。如果先对图像 I 施以变换，再进行卷积 f ，结果等同于对图像 I 的卷积施以变换。也就是说，如果图像中的目标发生了一定的位移，那么卷积输出的表达也会发生同等的位移。这一属性对于作用在一个相对小区域的算子来说是有用的。

下面以 CIFAR-10^[2] 数据集（如图 4-4 所示）为例说明卷积的优势。CIFAR-10 是一个由 10 个类别（包括：飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车）、60000 张彩色图像构成的分类任务数据集。输入图像尺寸为 $32 \times 32 \times 3$ （长 32、宽 32 及 3 个颜色通道），这样第一个隐层中的一个全连接神经元就包含 $32 \times 32 \times 3 = 3072$ 个权重（weight）。这个数字似乎并没有超出处理能力，但是当图像尺寸增加时，全连接结构显然无法控制网络规模。例如，输入尺寸为 $200 \times 200 \times 3$ 的图像，一个神经元就需要包含 $200 \times 200 \times 3 = 120000$ 个权重。而且每个隐层通常需要多个神经元，进而参数数量将快速上升。根据上面的分析可知，全连接结构对存储和运算资源消耗巨大，且数量庞大的参数会造成网络快速进入过拟合状态。

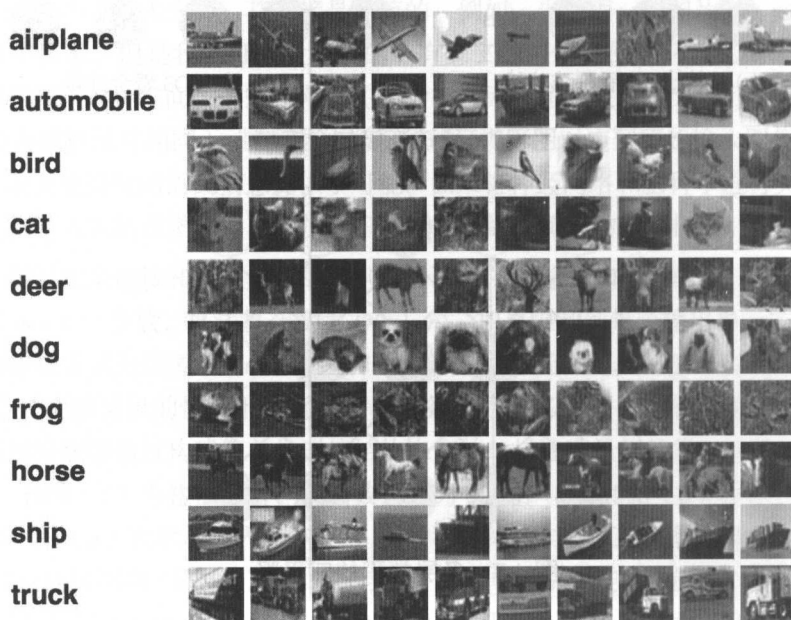


图 4-4 CIFAR-10 数据集

卷积网络的另一个优势在于以一种更符合逻辑的方式，利用并保持输入图像数据的三维结构：宽度、高度、深度（Width, Height, Depth）。注意这里的深度是指神经元的第三个维度，而不是整个网络的隐层数量。仍用 CIFAR-10，输入图像是一个尺寸为 $32 \times 32 \times 3$ 的三维数据。卷积层只与前层特征图的一个小区域相连。在卷积网络结构的末端，整张图像将被缩减为表示各个类别对应分数的向量。CIFAR-10 最后的输出层维度为 $1 \times 1 \times 10$ 。普通的三层神经网络如图 4-5 所示，而保持三维结构的卷积神经网络则如图 4-6 所示，图中输入层由一张图像构成，卷积的 3 个维度分别是宽度、高度和通道数（通常彩色图像可以拆解为 RGB 3 个通道）。

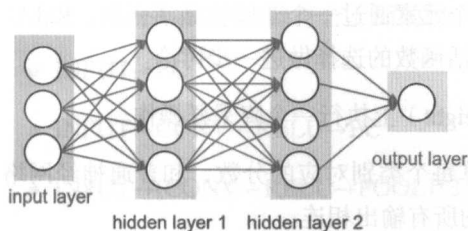


图 4-5 普通的三层神经网络

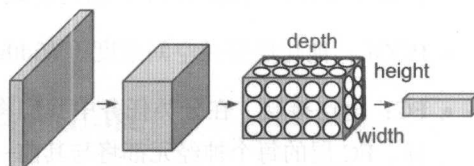


图 4-6 三维的卷积神经网络

4.3 卷积网络典型结构

4.3.1 基本网络结构

首先，我们回忆一下普通神经网络的结构。网络接收一个输入（Input，通常是一个一维向量），并由一系列的隐层（Hidden Layer）和输出层构成，每个神经元与上一层的所有神经元输出相连。隐层中的各个神经元完全独立，且不共享任何参数。网络中最后一个全连接层称为输出层（Output Layer），在分类任务中代表每个类别对应的分数。

同样，共享参数的卷积神经网络也具有类似的结构，这些不同的层相互组合就可以形成不同的网络结构。在深度学习的发展历程中，积累了很多非常经典的网络结构，比如 LeNet5、AlexNet、GoogLeNet、VGGNet16、ResNet 等，这些经典的网络结构往往代表了当时神经网络最强大的识别能力。

4.3.2 构成卷积神经网络的层

一个简单的卷积网络由一系列层构成，每层都将上一层的一组隐层输出通过一个可微函数产生一组新的隐层输出。一个典型的卷积网络可以由三种类型的层构成：卷积层（Convolutional Layer, CONV）配套 ReLU（Rectified Linear Unit, $\text{ReLU}(x) = \max(0, x)$ ）、池化层（Pooling Layer, POOL）和全连接层（Fully-Connected Layer, FC，和普通神经网络一致）。

像堆积木一样反复堆叠这些层便能构成一个卷积神经网络。这些层的具体作用^[3]如下。

- INPUT：图像的像素值作为输入。
- CONV：卷积层连接输入的一个小区域，并计算卷积核与对应的输入小区域之间的点乘作为输出。

- ReLU: 将 CONV 层中线性激活输出的每个元素通过一个非线性激活函数, ReLU 存在很多变种, 后面的章节还将对非线性激活函数的选择做进一步讨论。
- POOL: 池化层将在空间维度 (Width, Height) 上执行一个降采样操作。
- FC: 全连接层, 在分类任务中我们将计算每个类别对应的分数, 和普通神经网络一样, FC 层的每个神经元都将与其前一层的所有输出相连。

接下来, 我们用 CIFAR-10 的例子再复述一遍上述步骤。

- INPUT: 包含一张尺寸为 $32 \times 32 \times 3$ 的 RGB 图像。
- CONV: 假设当前 CONV 层包含 12 个卷积核, 那么通过 CONV 层后输出的尺寸变为 $32 \times 32 \times 12$ (当然, 这个尺寸还会受 stride 和 padding 等参数设置的影响, 详见后面章节中的讨论)。
- ReLU: 这一步不会改变特征图尺寸, 保持为 $32 \times 32 \times 12$ 。
- POOL: 以 2×2 小窗做降采样, 特征图尺寸降到 $16 \times 16 \times 12$ 。
- FC: 输出尺寸为 $1 \times 1 \times 10$, 表达 10 个类别分别对应的分数。

通过这样的方式, 卷积网络能够将原始图像转化为最终的多类别分数。在卷积网络中, 有的层有参数, 而有的层没有。具体的, CONV 和 FC 层不仅是关于输入的一个激活函数, 同时也具有参数: 各个神经元的权重 (weight) 和偏置 (bias)。这些参数通常采用梯度下降的方式习得。而 ReLU 和 POOL 则仅执行一个固定操作, 不具有参数。

4.3.3 网络结构模式

事实上, 很多卷积网络都能够被概括为相对固定的模式。

最常见的网络结构模式是若干 “CONV-ReLU” 堆叠, 之后跟随一个 POOL 层。多次重复这一结构, 直到图像在空间上被变换成一个比较小的尺寸。此后通常会转化成全连接的形式, 最后一个全连接层将作为输出层。

换句话说, 最常见的卷积神经网络结构可以概括为以下正则表达式^[3]:

$$\text{INPUT} \rightarrow [(\text{CONV} \rightarrow \text{RELU}) * N \rightarrow \text{POOL}] * M \rightarrow [\text{FC} \rightarrow \text{RELU}] * K \rightarrow \text{FC}$$

与普通正则表达式类似, 其中 ? 表示 0 或 1 次, * 表示重复, 后面的 N/M/K 表示对应重复的次数, 取值范围一般为: $0 < N \leq 3, M \geq 0, 0 < K \leq 3$ 。

这样，以下网络结构都符合上面的模式：

- INPUT→FC
- INPUT→CONV→RELU→FC
- INPUT→[CONV→RELU→POOL]*2→FC→RELU→FC
- INPUT→[CONV→RELU→CONV→RELU→POOL]*3→[FC→RELU]*2→FC

如图 4-7 所示为一个卷积神经网络结构示例^[3]，左侧的初始输入为图像的原始像素，右侧是最终在各个类别上判定的分数。其间从左到右的每一列表示网络中特定层的激活（特征图）。例子中使用的结构是一个小型的 VGGNet 网络，后面会进行详细描述。

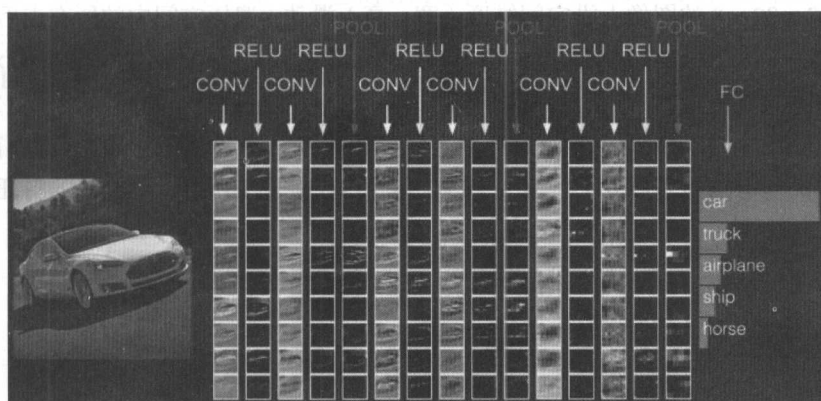


图 4-7 卷积神经网络结构示例

4.4 卷积网络的层

4.4.1 卷积层

卷积层是卷积网络的核心组成部分，包含了大部分繁重的计算工作。

1. 卷积层实现

卷积层的参数由一组可学习的卷积核（Filter）构成。每个卷积核在空间中都是小尺寸的（沿宽和高），但会穿过输入集整个深度。例如，卷积网络第一层的卷积核尺寸通常为

$5 \times 5 \times 3$ （宽、高各 5 像素，深度为彩色图像的 3 个通道）或 $3 \times 3 \times 3$ （宽、高各 3 像素，深度为彩色图像的 3 个通道）。

在前向传播过程中，我们在输入图像上沿宽和高的方向滑动各个卷积核（准确地讲，卷积），并在所有位置上分别计算卷积核和输入之间的点乘。当沿整个输入的宽和高方向滑动卷积核时，我们就会得到一个二维的激活映射（Activation Map），通常也称为特征图或特征映射（Feature Map），表示在每个空间位置上输入对于卷积核的响应。

直观地讲，网络将学习卷积核参数，使得在遇到某种视觉特征（如第一层某些方向上的边缘或某种颜色的斑点，或网络高层中的整个蜂窝状或轮状图案）时被激活。卷积层上的每个卷积核（如：例子 CIFAR-10 中 12 个卷积核）都会产生一个二维的激活映射，我们沿深度方向将这些激活映射排列起来，并将它们作为卷积层的输出。如图 4-8 所示为一个 $5 \times 5 \times 3$ 的卷积核在 $32 \times 32 \times 3$ 的图像上沿空间维度（宽、高）滑动，遍历空间中的所有点后便生成一个新的尺寸为 $28 \times 28 \times 1$ 的特征图。如图 4-9 所示为另一个 $5 \times 5 \times 3$ 的卷积核在 $32 \times 32 \times 3$ 的图像上沿空间维度（宽、高）滑动，遍历空间中的所有点后生成另一个新的尺寸为 $28 \times 28 \times 1$ 的特征图。如图 4-10 所示则是 6 个这样的卷积核在输入图像上沿空间维度（宽、高）滑动，遍历空间中的所有点后生成 6 个尺寸为 $28 \times 28 \times 1$ 的特征图，所以最终输出的特征图维度为 $28 \times 28 \times 6$ 。

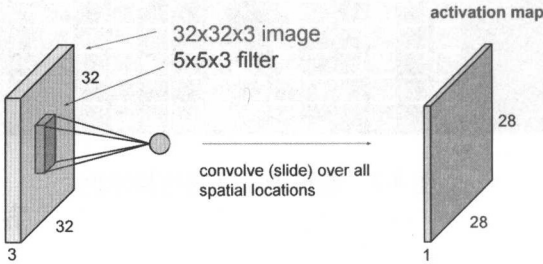


图 4-8 卷积层中的一个卷积核示例

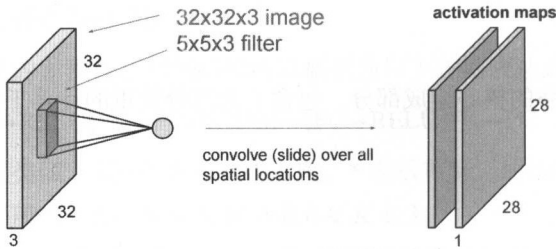


图 4-9 卷积层中的两个卷积核示例

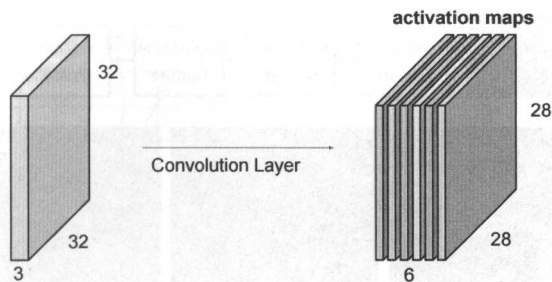


图 4-10 卷积层中的多个卷积核示例

在网络中堆叠 CONV-ReLU 结构。需要注意的是，卷积核的深度需要与输入的特征图的深度一致。如图 4-11 所示，第一个卷积层的卷积核尺寸为 $5 \times 5 \times 3$ ，其深度与输入图像 ($32 \times 32 \times 3$) 的深度一致；第二个卷积层的卷积核尺寸为 $5 \times 5 \times 6$ ，其深度就需要与第一个 CONV-ReLU 输出的特征图 ($28 \times 28 \times 6$) 的深度一致^[3]。

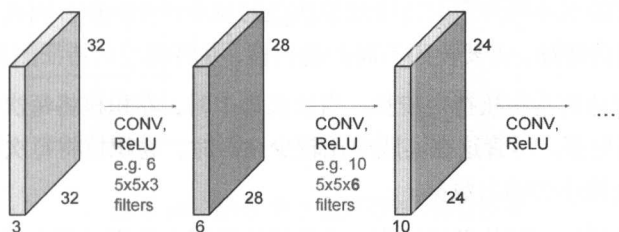


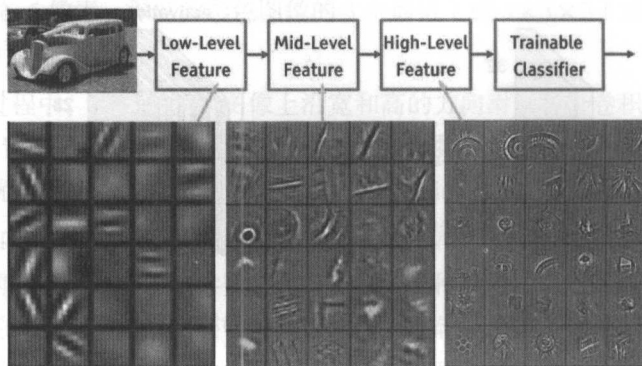
图 4-11 后一个卷积层的卷积核大小需要与前一个卷积层输出的维度一致

如图 4-12 所示，通过可视化各个卷积层输出的特征图，我们看到随着卷积网络的不断加深，特征图上的响应表现出的语义层次也在不断加深。最初的卷积层通常对图像中的边缘或色斑产生较强的响应，我们认为这个部分抽取的主要是低层特征（Low-Level Feature）。此后的卷积层在低层特征基础上产生的特征图开始出现一些具有一部分语义的图形或纹理。最后的卷积层倾向于对有明确语义的目标产生强响应，认为此时具有了抽取高层特征（High-Level Feature）的能力。

2. 空间排布

前面我们讨论了卷积层中每个神经元相对于输入的连接方式，接下来继续讨论神经元对于输出特征图有怎样的空间排布。

输出特征图的尺寸由三个超参数控制：深度（Depth）、步长（Stride）和零值填充（Zero-Padding）。

图 4-12 卷积层可视化^[4]

首先，输出特征图的深度是一个超参数。它对应于我们希望使用的卷积核的数量，每个卷积核都被训练为从图像中提取一些不同的信息。例如，第一个卷积层将原始图像作为输入，不同的卷积核可能对不同方向的边缘或带颜色的斑点产生响应。因此，对于同一个输入区域，为了提取不同的特征，需要使用不同的卷积核，并将响应的特征图排列起来作为输出。

其次，需要为滑动的卷积核指定步长。当步长为 1 时，卷积核将每次移动一个像素。当步长为 2（或取 3 或更多，尽管这在实践中比较少见）时，卷积核将每次移动两个像素，这将会产生空间尺寸比较小的输出数据。

最后，有时为了使用更深的卷积网络，不希望特征图在卷积过程中尺寸下降得太快，便会在输入的边缘填充零值。零值填充的大小同样是一个超参数。

那么通过卷积层后如何计算输出特征图的空间尺寸呢？

假设已知当前卷积层的输入图像尺寸（ W ）、卷积神经元可视野尺寸（ F ）、采用的步长（ S ）、边缘填充零值的数量（ P ），那么很容易得到输出特征图的尺寸计算公式为：

$$\frac{W - F + 2P}{S} + 1$$

我们可以通过下面的例子进一步验证这个结论^[5]。如图 4-13 所示为步长为 1 的卷积核，即卷积核每次沿高或宽移动一个像素，不进行边缘填充，由 $W = 7, F = 3, S = 1, P = 0$ ，计算得到输出尺寸为 5。如图 4-14 所示为步长为 2 的卷积核，即卷积核每次沿高或宽移动两个像素，我们直观地知道这时会得到一个相对小的输出尺寸。由 $W = 7, F = 3, S = 2, P = 0$ ，计算得到输出尺寸为 3。如图 4-15 所示为有填充时的情况，类似地， $W = 7, F = 3, P = 1$ ，当 $S = 1$ 时，输出尺寸为 7，可见，进行零值填充可以有效地保持输出特征图的尺寸和输入一致；当 $S = 2$ 时，输出尺寸为 4。

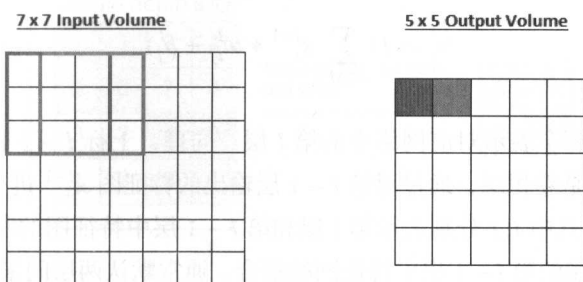


图 4-13 步长为 1 且不填充的卷积核

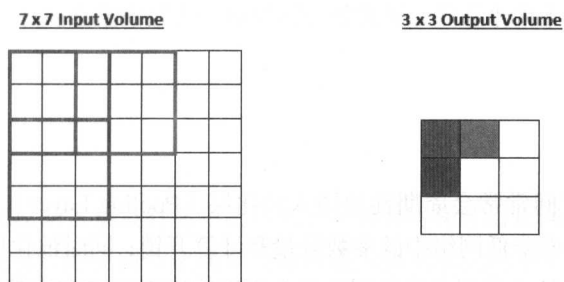


图 4-14 步长为 2 且不填充的卷积核

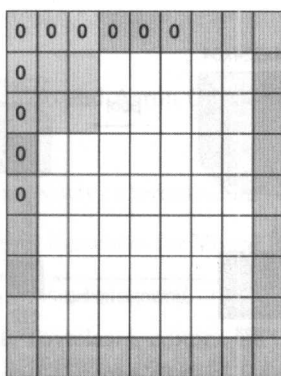


图 4-15 有填充时的卷积情况

3. 公式表达

这里我们进一步给出卷积层操作的公式表达^[3]:

$$x_j^l = f(\sum_{i \in M_j} x_i^{l-1} * w_{ij}^l + b_j^l)$$

上述公式中，上标 l 表示对应网络中的第 l 层。同理，上标 $l-1$ 表示对应网络中的第 $l-1$ 层。这里第 l 层是卷积层，通过对第 $l-1$ 层输出的特征图 x^{l-1} 进行卷积运算而获得本层的特征图输出 x_j^l ，其中 i, j 分别表示第 l 层和第 $l-1$ 层中特征图的序号。 M_j 表示与第 l 层第 j 个特征图相连接的第 $l-1$ 层中特征图的集合。通常默认两层间采用全连接，所以 M_j 包含第 $l-1$ 层的所有特征图。 w_{ij}^l 表示第 l 层第 j 个特征图对应第 $l-1$ 层第 i 个特征图输入的卷积核参数， b^l 表示偏置， w^l, b^l 为该卷积层的参数。 $*$ 表示卷积操作。

4.4.2 池化层

1. 池化层实现

在连续的卷积层之间常常会周期性地插入池化层（Pooling Layer）。池化层能够逐渐减小表达空间的尺寸，从而降低网络中的参数数量和计算开销；同时池化层也能起到控制过拟合的作用。池化层算子独立作用在特征图的每个深度维度上，并改变其空间中的尺寸。

如图 4-16 所示，输入特征图的深度为 64，空间尺寸为 224×224 。分别在每个深度上进行尺寸为 2×2 、步长为 2 的池化操作，对应得到 64 个空间尺寸为 112×112 的输出特征图。

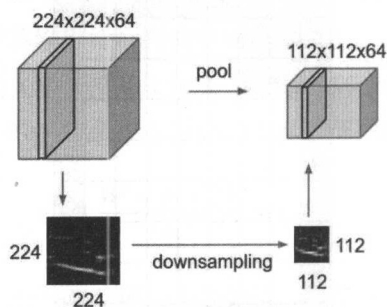
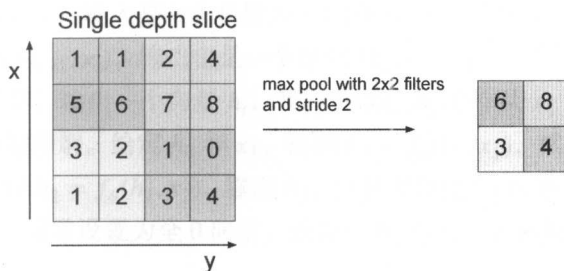


图 4-16 池化操作^[3]

最常见的池化操作是最大池化（Max Pooling），即取视野范围内的最大值。通常超参数有两种选择： $F=3, S=2$ 或 $F=2, S=2$ 。即步长为 2，池化窗口尺寸为 3 或 2。更大的池化窗口在实际中比较少见，因为过大的尺寸会给特征图信息带来严重的破坏。如图 4-17 所示为一个 $F=2, S=2$ 的最大池化示例。

图 4-17 尺寸为 2×2 、步长为 2 的最大池化示例^[3]

除了最大池化，一般的池化算子还有平均池化（Average Pooling），甚至 L2-Norm 池化（L2-Norm Pooling）。平均池化在过去一段时间比较常见，但近来由于最大池化被普遍证明有更好的效果而被取代。

2. 后向传播

最大池化操作的反向传播具有简单的形式：只需要将梯度沿正向传播过程中最大值的路径向下传递即可。池化层的正向传递通常会保留最大激活单元的下标，作为反向传播时梯度的传递路径。

参考文献

- [1] Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [2] The CIFAR-10 dataset. <https://www.cs.toronto.edu/~kriz/cifar.html>.
- [3] CS231n. Convolutional Neural Networks for Visual Recognition. <http://cs231n.github.io/>.
- [4] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In D. J. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, ECCV, volume 8689 of Lecture Notes in Computer Science, pages 818-833. Springer, 2014.
- [5] A Beginner's Guide To Understanding Convolutional Neural Networks. <https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/>.

5

循环神经网络

卷积神经网络适合处理单个目标类型的数据，在图像分类等领域获得广泛应用；而循环神经网络（Recurrent Neural Network，RNN）则适合处理序列类型的数据，在看图说话、语音识别、机器翻译等方面大放异彩。

本章将主要介绍 RNN 及其变种 LSTM、GRU、双向 RNN 的基本原理，然后以一个简单的语言模型为例讲解相关的代码实现。

5.1 循环神经网络简介

脑神经连接方式纵横交错，运行机制更是错综复杂，人们对其做了最大程度的简化，发明了人工神经网络用来模拟脑神经。输入的数值可以看作生物电信号，每个神经元接收的信号只有满足一定条件（可以看作激活函数）才会发射信号到下一个神经元，最后一个神经元告诉我们输入的数值具体是什么东西，这对应于图片分类场景。但是大脑不仅可以处理分类场景，还能处理一连串的输入。比如看电视，传入大脑的是一帧帧连续的图片，我们可以理解电视里发生的事情；别人说出的一个个字，传入大脑后，我们可以理解意图，进而也反馈一句话，这句话中的每个字也是有联系的。大脑可以处理这些连续的输入，是因为脑神经元之间的连接允许环的存在，神经元的输入可以是之前任何一个神经元的输出，而传统前馈神经网络是一个有向无环图，神经元的输入仅仅是上一层神经元的输出，于是人们根据脑神经的这种连接方式发明了循环神经网络。

如图 5-1 所示, RNN 的输入是一个长度为 T 的序列 $\{x_1, \dots, x_T\}$, 其中 x_t 表示一个向量, 内部使用函数 $h_t = f_\theta(h_{t-1}, x_t)$, 输出也是一个序列 $\{h_1, \dots, h_T\}$, 对于序列的每一个输入 x_t , 都用相同的 θ 进行计算, 输出一个向量 h_t , 我们可以把 h_t 看作从 x_1 到 x_t 的一个概括表示, 从图中可以获得直观的理解。给定 h_0 和 x_1 , 得到 $h_1 = f_\theta(h_0, x_1)$, 然后输出 h_1 , 并把 h_1 作为第二步的输入, 得到 $h_2 = f_\theta(h_1, x_2)$, 输出 h_2 , 并且同样把 h_2 作为第三步的输入。输入序列是从 x_1 开始的, h_0 通常设置为全 0 向量, 或者把 h_0 作为一种参数, 在训练过程中学习。

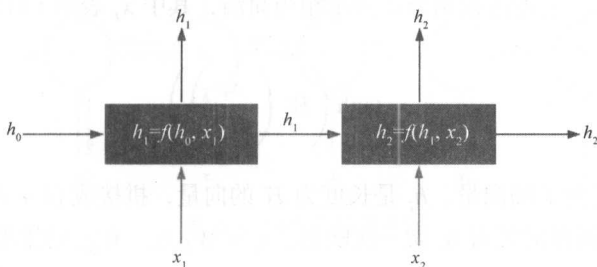


图 5-1 简单的 RNN 计算

需要注意的是, RNN 有时候也用作递归神经网络 (Recursive Neural Network) 的缩写。一般来说, 递归神经网络多用于自然语言处理中的序列和树结构学习, 是一种结构上递归的神经网络^[1], 而循环神经网络则是时间上线性递归的一种网络。但也有观点认为, 递归神经网络分为结构递归神经网络和时间递归神经网络, 其中时间递归神经网络就是循环神经网络。这种说法认为递归神经网络是包括循环神经网络的。在本章中, 如果没有特殊说明, RNN 都专指循环神经网络。

5.2 RNN、LSTM 和 GRU

普通的深度神经网络是由若干隐藏层“垂直”层叠而成的, 而 RNN 只有一个自连接的隐藏层, 这个隐藏层的输出作为下一时刻它的输入, 如果将 RNN 展开, 就可以看作若干隐藏层“水平”连接, 这些隐藏层共享同一套参数, 如图 5-2 所示。

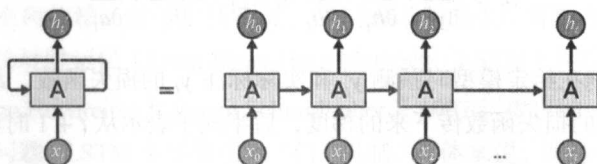


图 5-2 展开的 RNN 结构^[1]

每一时刻都有一个输入 x_t ，同时考虑之前所有的输入 x_1 到 x_{t-1} ，预测当前时刻的输出 h_t 。

RNN 的前向过程是按照时刻序列展开的，如果仅看某一时刻的前向过程，就是前一时刻输出一个向量，用其和当前时刻的输入向量进行拼接形成一个大向量，这个大向量经过一个隐藏层的输出，再经过激活函数，最终的输出向量作为当前时刻的输出；依次展开，当前时刻的输出向量再和下一时刻的输入向量拼接，拼接向量经过一个隐含层和激活函数的输出作为下一时刻的输入。上述过程用公式表示相当简洁，其中 x_t 表示 t 时刻的输入， h_t 表示 t 时刻的输出：

$$h_t = \tanh \left(W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} \right)$$

其中， x_t 是长度为 I 的向量， h_t 是长度为 H 的向量，拼接成 $(I + H)$ 的向量， W 的维度是 $H \times (I + H)$ 。通常需要对 h_t 做一次映射， $o_t = W_{ho}h_t$ ， W_{ho} 的维度是 $K \times H$ ，然后再接入 Softmax 进行 t 时刻的预测，即 $y'_t = \text{softmax}(W_{ho}h_t)$ 。为了下面表述方便，我们把上式写成： $h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1})$ 。通过公式可以看出，我们仅需要学习 $H(I + H) + HK$ 个参数，从理论上就可以处理任意长度的序列。然而，接下来我们将介绍 RNN 的后向过程，就会发现 RNN 在处理较长序列时，往往存在“梯度消失”和“梯度爆炸”的情况。

RNN 的训练算法通常有多种选择，比如 RTRL (Real Time Recurrent Learning)、BPTT (BackPropagation Through Time) 等，这里主要介绍 BPTT，因为 BPTT 更容易理解，并且计算效率更高。

就像普通的后向传播算法一样，BPTT 也是重复地使用链式法则。区别在于，对于 RNN 而言，损失函数不仅依赖于当前时刻的输出层，也依赖于下一时刻。除了输出层的参数 W_{ho} ， W_{xh} 和 W_{hh} 在计算更新梯度时都需要考虑当前时刻的梯度和下一时刻的梯度。对 W_{ho} 求导，其中 $z_t = W_{ho}h_t$ ：

$$\frac{\partial E_t}{\partial W_{ho}} = \frac{\partial E_t}{\partial y'_t} \cdot \frac{\partial y'_t}{\partial W_{ho}} = \frac{\partial E_t}{\partial y'_t} \cdot \frac{\partial y'_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial W_{ho}}$$

在对 W_{xh} 和 W_{hh} 求导之前，再定义两个变量： $a_t = W_{xh}x_t + W_{hh}h_{t-1}$ ， δ_t 表示在 t 时刻 a_t 接收到的梯度。

$$\delta_t = \frac{\partial E_t}{\partial y'_t} \cdot \frac{\partial y'_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial a_t} + \delta_{t+1} \cdot \frac{\partial a_{t+1}}{\partial h_t} \cdot \frac{\partial h_t}{\partial a_t}$$

E_t 表示当前时刻在给定模型时预测 y'_t 和实际标注 y_t 的损失函数， $E_t = E(y'_t, y_t)$ 。前半式子表示当前时刻 t 的损失函数传下来的梯度，后半式子表示从 $t + 1$ 时刻传过来的梯度。

需要注意的是, $\delta_{t+1} = 0$, 这是因为在最后一个时刻没有从下一时刻传过来的梯度。求出 δ_t 后, 就可以很容易求当前时刻对 W_{xh} 和 W_{hh} 的导数, 分别是 $\delta_t x_t^T$ 和 $\delta_t h_{t-1}^T$ (x_t^T 表示 x_t 的转置, h_{t-1}^T 同理)。可以看出, t 时刻的 E_t 求出的导数会依次传到 $t-1, t-2, \dots, 1$ 时刻, 如图 5-3 中的灰色箭头所示。

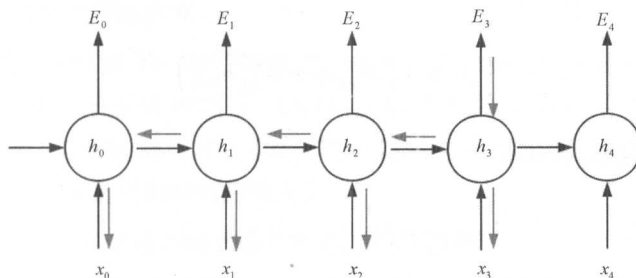


图 5-3 BPTT 示意图^[2]

最开始的时刻 1 得到时刻 t 损失函数 E_t 传来的导数是:

$$\frac{\partial E_t}{\partial y_t'} \cdot \frac{\partial y_t'}{\partial h_t} \cdot \left(\frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_2}{\partial h_1} \right) \cdot \frac{\partial h_1}{\partial a_1}$$

其中

$$\frac{\partial h_t}{\partial h_{t-1}} = W_{hh}^T \cdot (1 - \tanh(a_t))^2$$

$\tanh(a_t)$ 的导数是 $1 - \tanh(a_t)^2$, 范围是 $[0, 1]$, $\frac{\partial h_t}{\partial h_{t-1}}$ 通常是小于 1 的数, 所以上式括号中很多导数连乘的结果会非常接近于 0, 这就是“梯度消失”。如果忽略激活函数 Tanh 的导数, 上面时刻 1 获得时刻 t 传来的导数就是 $(W_{hh}^T)^t$, 即参数矩阵 W_{hh}^T 的 t 次幂, 设 W_{hh}^T 的特征值对角矩阵是 Λ , $(W_{hh}^T)^t$ 就和 Λ^t 相关, 如果某个特征值大于 1, $(W_{hh}^T)^t$ 就会很大, 可能出现“梯度爆炸”。如果某个特征值小于 1, $(W_{hh}^T)^t$ 就会很小, 也可能会出现“梯度消失”。

梯度爆炸相比梯度消失更容易出现, 从训练日志中如果发现很多数值出现 nan, 就说明很可能出现了梯度爆炸, 通常可以通过将梯度控制在一定范围内防止爆炸。除此之外, 在 RNN 的前向过程中, 开始时刻的输入对后面时刻的影响越来越小。这种前向和后向出现的问题我们称之为长距离依赖。对于上述情况, 我们可以用 ReLU 替换 Tanh 来解决, 但是更好的解决方案是用长短时记忆 (Long Short-Term Memory, LSTM) 网络替换 RNN。

LSTM 是由 Sepp Hochreiter 和 Jürgen Schmidhuber 于 1997 年提出的, 解决了 RNN 训练过程中的长距离依赖问题。LSTM 主要靠引入“门”机制, 具体来说, 就是引入了“输入门”“遗忘门”和“输出门”以及相关的变量, 具体公式如下:

(1) 输入门 (Input Gate)

$$i_t = \sigma(W_{xh}^i x_t + W_{hh}^i h_{t-1})$$

(2) 遗忘门 (Forget Gate)

$$f_t = \sigma(W_{xh}^f x_t + W_{hh}^f h_{t-1})$$

(3) 输出门 (Output Gate)

$$o_t = \sigma(W_{xh}^o x_t + W_{hh}^o h_{t-1})$$

(4) 输入门相关状态值

$$g_t = \tanh(W_{xh}^g x_t + W_{hh}^g h_{t-1})$$

(5) 单元状态值 (Cell State)

$$c_t = c_{t-1} \circ f_t + g_t \circ i_t$$

(6) 当前隐藏状态 (Hidden State) 的输出

$$h_t = \tanh(c_t) \circ o_t$$

其中, \circ 表示按元素的乘积。

从公式上看非常复杂,但是理清楚之后就会发现,就是在 RNN 的基础上套了三个门并引入了一个状态值。输入和 RNN 相比,除了当前时刻的输入和前一时刻的输出,就多了一个前一时刻的状态值。同样,输出也多了一个当前时刻的状态值。公式中的 $g_t = \tanh(W_{xh}^g x_t + W_{hh}^g h_{t-1})$ 就是基本的 RNN 逻辑。

i, f, o 分别是输入门、遗忘门和输出门,它们有形式一样的公式、共同的输入,只是参数不一样。它们最外面的函数是 Sigmoid,输出值的范围是 $[0, 1]$, 用一个向量和它们进行元素相乘,就可以看作有个门来控制这个向量“通过”的程度。通过当前输入和前一时刻的输出计算出当前时刻的状态值,而输入门控制了新状态值通过的程度,遗忘门控制了前状态遗忘的程度,输出门决定了当前状态值可以被输出的程度。每个门的维度、状态值的维度和隐藏层输出的维度都是一样的。

g 是通过当前输入和前一时刻的输出计算的候选状态值。在 RNN 中就直接拿 g 作为输出，但是在 LSTM 中需要输入门来控制 g 的值。

c_t 是状态值，它是遗忘门乘以前状态值和输入门乘以候选状态值 g 两个乘积的和。直观上理解就是，我们选择前状态的一些值和候选状态的一些值组成了目前的状态。当前状态值乘以输出门就是当前时刻的输出 h_t 。

RNN 可以认为是 LSTM 的一种特殊形式，将 LSTM 的输入门都设为 1，遗忘门都设为 0，输出门都设为 1，就几乎变成 RNN 了。LSTM 正是通过这些门解决了长距离依赖问题，通过训练这些门的参数，LSTM 就可以自主决定当前时刻的输出是依赖于前面的较早时刻，还是前面的较晚时刻，抑或是当前时刻的输入。

图 5-4 和图 5-5 可以形象地表明 LSTM 相比 RNN 的优势。在图 5-4 中，节点阴影的深浅程度表明节点时刻 1 对其他时刻的影响程度，颜色越深，影响越大。从图中可以看出，后面节点受时刻 1 的影响越来越小，时刻 6 和 7 已经忘记了时刻 1 的输入。

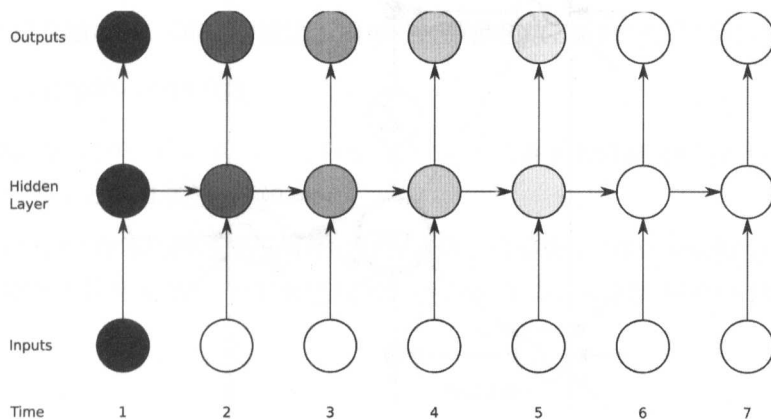


图 5-4 RNN 中的长距离信息消失问题^[3]

在图 5-5 中，输入门、遗忘门和输出门分别在隐藏节点的下面、左面和上面。空心小圆圈“o”表示门是打开的，“-”表示门是关闭的。可以看到，如果时刻 2 到时刻 6 的输入门都是关闭的，那么时刻 1 对时刻 6 的影响并没有减弱。

Alex Graves 在 LSTM 基础上提出了 peephole 的概念，图 5-6 中的虚线就是 peephole，直观上看，就是三个门都加了一个窥探上一个状态值的机会。

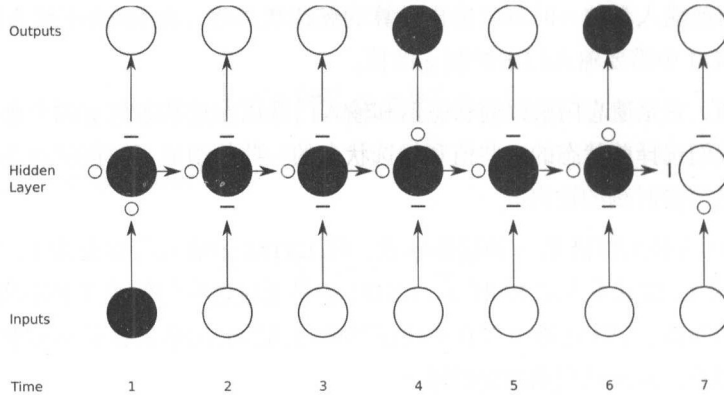


图 5-5 LSTM 实现信息的长距离有效传播^[3]

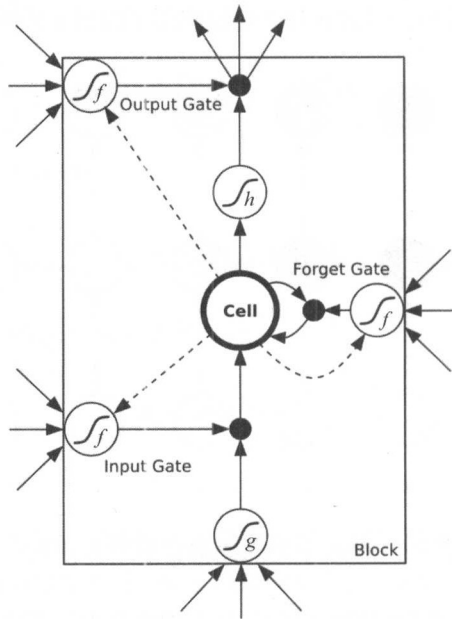


图 5-6 LSTM 中含有一个细胞单元的内存块

就是在三个门的计算中加入了上一时刻的状态值：

$$i_t = \sigma(W_{xh}^i x_t + W_{hh}^i s_{t-1} + W_{ch}^i c_{t-1})$$

$$f_t = \sigma(W_{xh}^f x_t + W_{hh}^f s_{t-1} + W_{ch}^f c_{t-1})$$

$$o_t = \sigma(W_{xh}^o x_t + W_{hh}^o s_{t-1} + W_{ch}^o c_{t-1})$$

其他部分不变。加入 peephole 之后，由于增强了前一时刻状态值的影响，所以效果也会得到提升。

GRU^[4] 是 Cho 在 2014 年提出的一种 LSTM 简化版本，先列出 GRU 的计算公式：

$$\begin{aligned} z_t &= \sigma(W_{xh}^z x_t + W_{hh}^z h_{t-1}) \\ r_t &= \sigma(W_{xh}^r x_t + W_{hh}^r h_{t-1}) \\ \tilde{h}_t &= \tanh(W_{xh}^h x_t + W_{hh}^h (h_{t-1} \circ r_t)) \\ h_t &= (1 - z_t) \circ \tilde{h}_t + z_t \circ h_{t-1} \end{aligned}$$

从公式上看，GRU 相比 LSTM 少了状态值和一个门。GRU 只有两个门：一个重置门 r 和一个更新门 z 。如果将 $\tilde{h} = \tanh$ 看作一个黑盒的话，重置门决定了前时刻对黑盒输入值的影响程度，而更新门决定了前时刻对从黑盒输出值的影响程度。如果将重置门设为 1，更新门设为 0，GRU 就变成了标准的 RNN。GRU 和 LSTM 的不同之处在于：

- GRU 没有输出门，GRU 的输出不用再经过激活函数，GRU 的参数更少。
- GRU 没有维护一个状态值。
- 在 GRU 的第四个公式中， $(1 - z) \circ \tilde{h}$ 中的 $1 - z$ 代替了 LSTM 中的输入门， $z \circ h_{t-1}$ 中的 z 代替了 LSTM 的遗忘门。

LSTM 和 GRU 在不同的场景下对比时各有胜负，但是由于 GRU 的参数更少，如果训练数据较少，则推荐使用 GRU；如果训练数据很多，就可以更充分地训练 LSTM 的参数，效果也更好些。

5.3 双向 RNN

上文介绍的 RNN 或者 LSTM 只是单向的，即位置（或时刻） i 的状态值只与从位置 0 到位置 i 的输入有关，与从位置 $i + 1$ 到结束的输入都没有关系，也就是只有“上文”。如何让每个位置的状态值都有“上下文”信息呢？我们可以让 RNN 反向再计算一遍，即从输入的结束位置走向开始位置，每个位置的状态值就是正向 RNN 和反向 RNN 在当前位置状态值的拼接。我们将这种 RNN 称为双向 RNN，如图 5-7 所示。

显而易见，双向 RNN 比单向 RNN 的表达能力更强。在 OCR、机器翻译和语音识别等领域，如果输入序列是确定的，则可以使用双向 RNN，最终的效果往往比单向 RNN 要好很多。然而，双向 RNN 的耗时也是单向 RNN 的两倍，在实际应用中要综合考虑效果和性能。

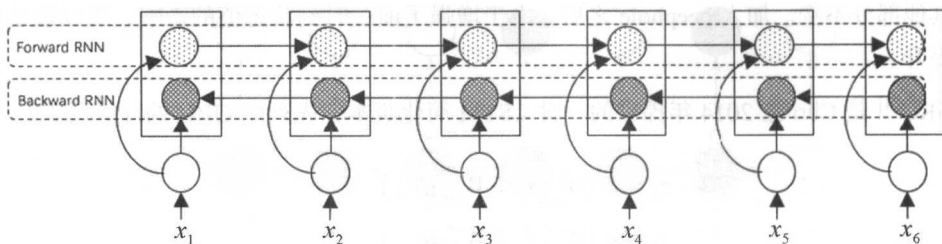


图 5-7 双向 RNN

5.4 RNN 语言模型的简单实现

语言模型在很多领域都发挥着至关重要的作用，比如中文分词、统计机器翻译、拼写检查、语音识别等领域。直观上讲，语言模型就是预测句子中的每个词出现的概率，这个概率是通过当前词的前面所有词计算的。我们通常采用 n -gram 统计语言模型，即假设句子中的每个词只与其前 $n-1$ 个词有关。但是，采用 n -gram 统计出来的语言模型还是存在数据稀疏的问题，目前通常采用基于 RNN 的语言模型来避免这个问题。

RNN 语言模型的输入就是一个句子，在理论上不限制句子的长度，但是考虑到模型收敛的速度，在实际训练过程中会设置句子的最大长度，超过的部分将会被删除。设句子的当前位置为 t ，给 RNN 语言模型输入从句子开始 0 到 $t-1$ 位置共 t 个词，预测当前位置 t 最应该出现什么词。比如有一个句子：“今天天气很晴朗”，经过分词后变成“今天 天气 很 晴朗”，首先在给定“今天”的情况下，RNN 预测出“天气”；然后再给定“今天 天气”，RNN 预测出“很”；再给定“今天 天气 很”，RNN 预测出“晴朗”。

对于每个词都对应一个向量，我们通常称之为词嵌入 (Embedding)。如图 5-8 所示，依次将句子中的每个词对应的向量放入 RNN 中，经过一个隐藏层，再经过一个输出层，RNN 输出下一个位置对应的词的分布概率。

现在，我们使用 Python 实现简单的 RNN 语言模型（以下程序参考了文献 [5]）。首先，定义 6 种参数。

```
embedding = numpy.random.randn(vocab_size, hidden_size)
w_x = numpy.random.randn(hidden_size, hidden_size) #embedding映射到隐藏层
w_h = numpy.random.randn(hidden_size, hidden_size) #h_{t-1}映射到隐藏层
w_y = numpy.random.randn(vocab_size, hidden_size) #h_t映射到输出层
b_h = numpy.zeros((hidden_size, 1)) #隐藏层bias
b_y = numpy.zeros((vocab_size, 1)) #输出层bias
```

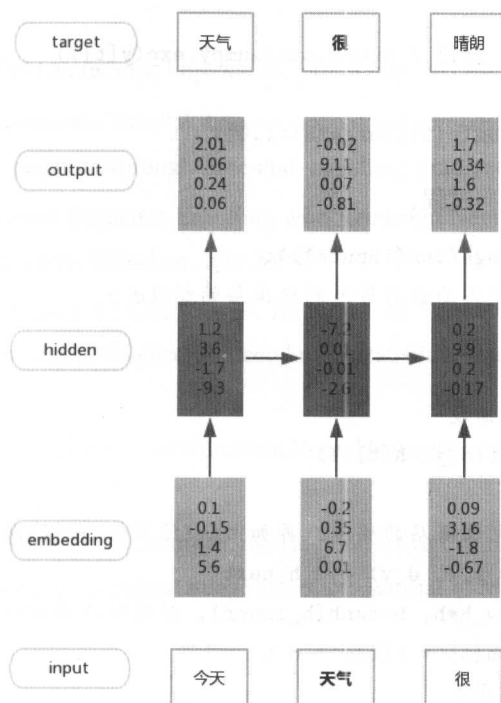


图 5-8 RNN 语言模型示例

RNN 语言模型的前向过程：

```
#line是一个分过词的句子，sentence2id将词变成id，sentence就是词id的序列
sentence = sentence2id(line)
#词id从0到vocab_size-1，结束标记的id是vocab_size，每句话添加一个结束标记
sentence.append(vocab_size)
#构造输入和输出
inputs = sentence[0:len(sentence)-1]
targets = sentence[1:len(sentence)]
for t in xrange(len(inputs)):
    x[t] = embedding[inputs[t]].reshape(hidden_size,1)
    #隐藏层
    h[t] = numpy.tanh(numpy.dot(w_x, x[t]) + numpy.dot(w_h, h[t-1]) + b_h)
    #输出层
```

```

y[t] = numpy.dot(w_y, h[t]) + b_y
#Softmax
p[t] = numpy.exp(y[t]) / numpy.sum(numpy.exp(y[t]))
#交叉熵损失
loss += - numpy.log(p[t][targets[t],0])

```

RNN 语言模型的后向过程:

```

for t in reversed(xrange(len(inputs))):
    #Softmax和交叉熵损失函数的传递到输出层的梯度d_y
    d_y = ps[t]
    d_y[targets[t]] -= 1
    #输出层参数的更新梯度
    d_w_y += numpy.dot(d_y, h[t].T)
    d_b_y += d_y
    #t位置输出层传递到隐藏层的梯度，再加上t+1位置传递到隐藏层的梯度
    d_h = numpy.dot(w_y.T, d_y) + d_h_next
    #h_inner = w_x*x+w_h*h, h=tanh(h_inner)，经过激活函数传递的梯度
    d_h_inner = (1 - h[t] * h[t]) * d_h
    #隐藏层参数的更新梯度
    d_b_h += d_hrow
    d_w_x += numpy.dot(d_h_inner, x[t].T)
    d_w_h += numpy.dot(d_h_inner, h[t-1].T)
    #embedding层参数的更新梯度
    d_embedding_t = d_embedding[inputs[t]].reshape(hidden_size,1)
    d_embedding_t += numpy.dot(w_x.T, d_h_inner)
    #传递到t-1位置的梯度
    d_h_next = numpy.dot(w_h.T, d_h_inner)

```

以上程序只能跑很小的数据集，要使用实际数据来训练语言模型的话，首先得采用 mini-batch 的模型训练。最重要的是不能采用 Softmax，因为实际数据集的词汇表会非常大，通常在十几万到百万之间，而 Softmax 必须要加和所有词汇表的得分来计算归一化系数，会非常占内存，并且计算非常慢。在这种情况下，我们通常采用 NCE(Noise Contrastive Estimation)^[6] 替代 Softmax，这里不再展开叙述，有兴趣的读者可以研究相关论文。

参考文献

- [1] Understanding LSTM Networks. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.
- [2] Recurrent Neural Networks Tutorial. <http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/>.
- [3] A. Graves. Supervised Sequence Labelling with Recurrent Neural Networks. Dissertation. Technische Universität München, München, July 2008.
- [4] K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. EMNLP, 2014.
- [5] The Unreasonable Effectiveness of Recurrent Neural Networks. <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.
- [6] Aapo Hyvarinen Michael U. Gutmann. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. Journal of Machine Learning Research, 12:307-361, 2012.

6

深度学习优化算法

在传统机器学习算法的实践中，优化总是重头戏，也是最考验功底的部分。深度学习得益于后向传播的有效方式，往往普通的随机梯度下降优化方法就能取得不错的训练效果，优化的重要性相比传统机器学习要弱一些，大部分从业者主要聚焦于应用或模型创新，而优化部分更多的工作只是调参。

实际上，深度学习优化方面的研究非常多，很多方法也非常有效，尤其在数据量较大的时候，所以有必要掌握一些常见的优化算法，本章将带领大家一起学习这些方法。

6.1 SGD

随机梯度下降（Stochastic Gradient Descent, SGD）每次从训练样本中随机抽取一个样本计算 loss 和梯度并对参数进行更新，由于每次不需要遍历所有的数据，所以迭代速度快；但是这种优化算法比较弱，往往容易走偏，反而会增加很多轮迭代。随机梯度下降有时可以用于在线学习（Online Learning）系统，可使系统快速学到新的变化。

与随机梯度下降相对应的还有批量梯度下降（Batch Gradient Descent, BGD），每次使用整个训练集计算梯度，这样计算的梯度比较稳定，相比随机梯度下降不那么容易震荡；但是因为每次都需要更新整个数据集，所以批量梯度下降算法非常慢而且无法放在内存中计算，更无法应用于在线学习系统。

介于随机梯度下降和批量梯度下降之间的是小批量梯度下降（Mini-Batch Gradient Descent），即每次随机抽取 m 个样本，以它们的梯度均值作为梯度的近似估计。

在深度学习中常说的随机梯度下降通常是指小批量梯度下降，本书后面部分如果没有特殊提及，随机梯度下降都是指小批量梯度下降。

随机梯度下降算法的具体训练步骤如算法 6-1 所示。其中 $f(x^{(i)})$ 表示第 i 个样本的预测值， $L(f(x^{(i)}), y)$ 表示第 i 个样本的损失函数， $\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)})$ 为 m 条数据的平均损失。

算法 6-1 随机梯度下降算法

参数：学习率 η

初始化： θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度： $g(\theta) = \frac{\partial(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\partial \theta}$

更新参数： $\theta = \theta - \eta \times g(\theta)$

end while

为了使随机梯度下降获得较好的性能，学习率 η 需要取值合理并根据训练过程动态调整。如果学习率过大，模型就会收敛过快，最终离最优值较远；如果学习率过小，迭代次数就会很多，导致模型长时间不能收敛。

6.2 Momentum

动量（Momentum）是来自中学物理力学中的一个概念，是力的时间积累效应的度量。动量^[1]的方法在随机梯度下降的基础上，加上了上一步的梯度：

$$m_t = \gamma m_{t-1} + g(\theta)$$

$$\theta = \theta - \eta m_t$$

其中 γ 是动量参数且 $\gamma \in [0, 1]$ 。动量的优化方法也可以写为如下形式：

$$v_t = \gamma v_{t-1} + \eta \times g(\theta)$$

$$\theta = \theta - v_t$$

区别在于学习率 η 的位置。如果对公式进行展开，不难发现两者是完全等价的，为了保持一致，本书采用第一种写法。

相比随机梯度下降，动量会使相同方向的梯度不断累加，而不同方向的梯度则相互抵消，因而可以在一定程度上克服 Z 字形的震荡，更快到达最优点。具体的算法细节如算法 6-2 所示。

算法 6-2 带动量的随机梯度下降算法

参数：学习率 η ，动量 μ

初始化： θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度： $g(\theta) = \frac{\partial(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\partial \theta}$

更新参数：

$$(1) m_t = \gamma m_{t-1} + g(\theta)$$

$$(2) \theta_{t+1} = \theta_t - \eta m_t$$

end while

6.3 NAG

Nesterov 加速梯度 (Nesterov Accelerated Gradient, NAG)^[2] 与动量类似，也是考虑最近的梯度情况，但是 NAG 相对超前一点，它先使用动量 m_t 计算参数 θ 下一个位置的近似值 $\theta + \eta m_t$ ，然后在近似位置上计算梯度：

$$m_t = \gamma m_{t-1} + g(\theta_t - \eta \gamma m_{t-1})$$

$$\theta_{t+1} = \theta_t - \eta m_t$$

NAG 与动量法的具体区别如图 6-1 所示。从图中可以看出，NAG 算法会计算本轮迭代时动量到达位置的梯度，可以说它计算的是“未来”的梯度。如果未来的梯度存在一定的规律，那么这些梯度就会有更好的利用价值。如果采用这种计算方式，梯度计算采用的是点 A，前向计算采用的是原始的点 O，这种不统一带来了额外的计算开销。

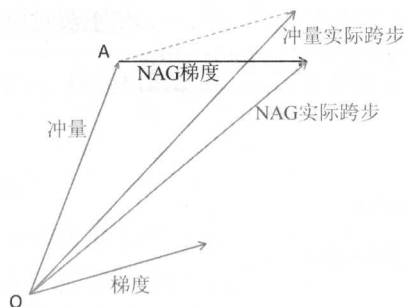


图 6-1 NAG 与动量法的区别

当然，在实际计算过程中，比如开源框架 Caffe，为了前向、后向计算统一，引入了以下变量：

$$\hat{\theta}_t = \theta_t - \eta \gamma m_{t-1}$$

$$\hat{\theta}_{t+1} = \theta_{t+1} - \eta \gamma m_t$$

将上面两个公式代入，就可以得到：

$$m_t = \gamma m_{t-1} + g(\hat{\theta}_t)$$

$$\hat{\theta}_{t+1} + \eta \gamma m_t = \hat{\theta}_t + \eta \gamma m_{t-1} - \eta m_t$$

将上面的第一个公式代入第二个公式，就可以得到：

$$\hat{\theta}_{t+1} + \eta \gamma m_t = \hat{\theta}_t + \eta(m_t - g(\hat{\theta}_t)) - \eta m_t$$

整理得到：

$$\hat{\theta}_{t+1} = \hat{\theta}_t + \eta m_t - \eta g(\hat{\theta}_t) - \eta m_t - \eta \gamma m_t$$

$$\hat{\theta}_{t+1} = \hat{\theta}_t - \eta g(\hat{\theta}_t) - \eta \gamma m_t$$

这样梯度计算和前向计算不一致的问题就得到了解决。

NAG 对应的算法如算法 6-3 所示。

算法 6-3 Nesterov 加速梯度算法

参数：学习率 η ，动量衰减率 γ

初始化： θ

while 停止条件未满足 do

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

$$\text{计算梯度: } g(\hat{\theta}_t) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta \hat{\theta}_t}$$

更新参数:

$$(1) \quad m_t = \gamma m_{t-1} + g(\hat{\theta}_t)$$

$$(2) \quad \hat{\theta}_{t+1} = \hat{\theta}_t - \eta g(\hat{\theta}_t) - \eta \gamma m_t$$

end while

6.4 Adagrad

Adagrad^[3] 是一种自适应的梯度下降算法,它能够针对参数更新的频率调整它们的更新幅度——对于更新频繁且更新量大的参数,适当减小它们的步长;对于更新不频繁的参数,适当增大它们的步长。这种方法的思想很适合一些数据分布不均匀的任务,比如对于一些自然语言处理问题,有些频繁出现的单词会给予更频繁的更新,有些不频繁出现的单词则更难进行参数更新。对于这样的问题,使用 Adagrad 可以更好地平衡参数更新的量,使模型的表现更好。

它的具体更新方法是在之前梯度下降法的基础上增加一个梯度的积累项作为分母,之前梯度下降法的参数更新公式为:

$$\theta_{t+1} = \theta_t - \eta g_t$$

而 Adagrad 变成:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} \odot g_t$$

其中 \odot 表示两个向量元素级 (Element-wise) 的乘法, G_t 就是 Adagrad 增加的内容。它是所有轮迭代的梯度平方和:

$$G_t = \sum_{k=1}^t g_k^2$$

从公式可以看出,加入这一项之后,参数的更新确实得到了一定的控制。对于经常更新的参数, G_t 项的数值会比较大,因而它的参数更新量会得到控制;对于不常更新的参数,由于 G_t 项的数值比较小,它的参数更新量会变大。完整的算法如算法 6-4 所示。

算法 6-4 Adagrad 自适应梯度法

参数: 学习率 η , 微小量 ϵ , 梯度积累量 G

初始化: θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度: $g(\theta) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta \theta}$

更新梯度积累量: $G_t = G_{t-1} + g^2(\theta)$

更新参数: $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g(\theta)$

end while

从算法中可以看出, Adagrad 方法存在一点缺陷。如果模型的参数数值保持稳定, 那么参数的梯度值总体不会有太大的波动, 而分母上的梯度积累项一直在累积, 因此分母会不断变大, 因此从梯度的趋势上分析, 梯度总体上会不断变小。虽然在实际训练中一般也会将学习率调小, 但两者变小的程度不同, 因此 Adagrad 可能会出现更新量太小而不易优化的情况。

6.5 RMSProp

RMSProp^{[4][5]} 利用滑动平均的方法来解决 Adagrad 算法中的问题。它的思路是让梯度积累值 G 不要一直变大, 而是按照一定的比率衰减, 这样其含义就不再是梯度的积累项了, 而是梯度的平均值:

$$\begin{aligned} G_{t+1} &= \gamma G_t + (1 - \gamma) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \end{aligned}$$

因为此时的 G 更像是梯度的平均值甚至期望值, 因此在很多文献中会将 G 写成 $E[g^2]$ 。

RMSProp 算法如算法 6-5 所示。

算法 6-5 RMSProp 算法

参数: 学习率 η , 微小量 ϵ , 梯度积累量 G , 梯度积累量衰减率 γ

初始化: θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

$$\text{计算梯度: } g(\theta) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta \theta}$$

$$\text{更新梯度积累量: } G_t = \gamma G_{t-1} + (1 - \gamma) g^2(\theta)$$

$$\text{更新参数: } \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g(\theta)$$

end while

6.6 Adadelta

Adadelta^[6] 算法的思想和 RMSProp 算法比较接近, 不过 Adadelta 考虑了一些更新量“单位”的问题。对比 Adagrad 算法和梯度下降法的更新公式, 可以得到如表 6-1 所示的对比结果。

表 6-1 Adagrad 算法和梯度下降法的更新公式对比

方法	更新公式
梯度下降	$\theta_{t+1} = \theta_t - \eta g_t$
Adagrad	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$

可以看出 Adagrad 算法和梯度下降法相比多出来一个项目, 这样更新量的“单位”就和之前不同了。为了让“单位”匹配, Adadelta 选择在分子上再增加一个项目, 于是方法的概念公式变成:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \Delta \theta_t \\ \Delta \theta_t &= -\frac{\text{RMS}[\Delta \theta]_{t-1}}{\text{RMS}[g]_t} \odot g_t \end{aligned}$$

其中 RMS 表示 Root Mean Square, 也就是“均方根”的意思。分母中的 $\text{RMS}[g]_t$ 展开与 RMSProp 相同:

$$\text{RMS}[g]_t = \sqrt{\gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 + \epsilon}$$

分子部分也采用类似的方法，展开后得到：

$$\begin{aligned}\text{RMS}[\Delta\theta]_t &= \sqrt{E[\Delta\theta^2]_t + \epsilon} \\ E[\Delta\theta^2]_t &= \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta_t^2 \\ \Delta\theta_t &= -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} \odot g_t\end{aligned}$$

最终的算法如算法 6-6 所示。

算法 6-6 Adadelta 算法

参数：学习率 η ，微小量 ϵ ，梯度积累量 G ，衰减率 γ

初始化： θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度： $g(\theta) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta\theta}$

更新梯度积累量： $G_t = \gamma G_{t-1} + (1 - \gamma)g^2(\theta)$

计算参数更新量： $\Delta\theta = -\frac{\sqrt{\Delta_{t-1} + \epsilon}}{\sqrt{G_t + \epsilon}} \odot g(\theta)$

更新参数相关积累量： $\Delta_t = \gamma \Delta_{t-1} + (1 - \gamma)\Delta\theta_t^2$

更新参数： $\theta_{t+1} = \theta_t + \Delta\theta$

end while

6.7 Adam

Adam 算法的全称为 Adaptive Moment Estimation^[7]，这种方法结合了上面提到的两类算法：基于动量的算法和基于自适应学习率的算法。基于动量的算法有动量法和 NAG 法，这两种方法都基于历史的梯度信息进行参数更新。基于自适应学习率的算法有 Adagrad、RMSPprop、Adadelta，它们通过计算梯度的积累量来调整不同参数的更新量。Adam 算法记录了梯度的一阶矩（梯度的期望值）和二阶矩（梯度平方的期望值）：

$$\begin{aligned}m_t &= \beta_1 m_{t-1} + (1 - \beta_1)g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)g_t^2\end{aligned}$$

为了确保两个梯度积累量能够良好地估计梯度的一阶矩和二阶矩，两个积累量还需要乘以一个偏置纠正的系数：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

然后再使用两个积累量进行参数更新：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$$

完整的算法如算法 6-7 所示。

算法 6-7 Adam 算法

参数：学习率 η ，微小量 ϵ ，一阶矩 \hat{m}_t ，二阶矩 \hat{v}_t ，衰减率 β_1 、 β_2

初始化： θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度： $g(\theta) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta\theta}$

更新一阶矩： $m_t = \beta_1 m_{t-1} + (1 - \beta_1)g(\theta)$

更新二阶矩： $v_t = \beta_2 v_{t-1} + (1 - \beta_2)g^2(\theta)$

纠正一阶矩： $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$

纠正二阶矩： $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$

计算参数更新量： $\Delta\theta = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$

更新参数： $\theta_{t+1} = \theta_t + \Delta\theta$

end while

6.8 AdaMax

AdaMax 算法^[7]主要针对 Adam 算法进行了修改,而修改的位置在二阶矩 v_t 这里。AdaMax 将二阶矩修改为无穷矩,这样在数值上更加稳定:

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned}$$

将 v_t 替换为 u_t 后,最终的更新变为:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \odot \hat{m}_t$$

此时的无穷矩估计不再是偏的,因此也不需要再做纠正。最终的算法如算法 6-8 所示。

算法 6-8 AdaMax 算法

参数: 学习率 η , 微小量 ϵ , 一阶矩 \hat{m}_t , 二阶矩 \hat{v}_t , 衰减率 β_1 、 β_2

初始化: θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度: $g(\theta) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta \theta}$

更新一阶矩: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g(\theta)$

更新无穷矩: $u_t = \max(\beta_2 u_{t-1}, |g(\theta)|)$

纠正一阶矩: $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$

计算参数更新量: $\Delta \theta = -\frac{\eta}{\sqrt{\hat{u}_t} + \epsilon} \odot \hat{m}_t$

更新参数: $\theta_{t+1} = \theta_t + \Delta \theta$

end while

6.9 Nadam

6.8 节的 AdaMax 算法修改了二阶矩的估计值,本节的算法则修改了一阶矩的估计值,将 Nesterov 算法和 Adam 算法结合起来,形成了 Nadam (Nesterov-accelerated Adaptive Moment Estimation) 算法^[8]。

在 6.3 节我们已经看到了 NAG 算法的公式：

$$\begin{aligned} m_t &= \gamma m_{t-1} + \eta g_t \\ \theta_{t+1} &= \theta_t - (\gamma m_t + \eta g_t) \end{aligned}$$

而 6.7 节 Adam 算法的更新公式可以展开为：

$$\begin{aligned} \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t \\ &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \left(\frac{\beta_1 m_{t-1} + (1 - \beta_1) g_t}{1 - \beta_1^t} \right) \\ &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \left(\beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right) \\ &= \theta_t - \left(\frac{\eta \beta_1}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_{t-1} + \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \left[\frac{(1 - \beta_1)}{1 - \beta_1^t} g_t \right] \right) \end{aligned}$$

可以看出，公式的形式和 NAG 算法很相近，为了体现 Nesterov 的效果，只需要将公式中的 m_{t-1} 修改为 m_t 即可。Nadam 算法如算法 6-9 所示。

算法 6-9 Nadam 算法

参数：学习率 η ，微小量 ϵ ，一阶矩 \hat{m}_t ，二阶矩 \hat{v}_t ，衰减率 β_1 、 β_2

初始化： θ

while 停止条件未满足 **do**

从训练数据中抽取 m 条数据 $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$ 及对应的标签 $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$

计算梯度： $g(\theta) = \frac{\delta(\frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}), y^{(i)}))}{\delta \theta}$

更新一阶矩： $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g(\theta)$

更新一阶矩的 Nesterov 加速值： $\tilde{m}_t = \beta m_t + (1 - \beta_1) g_t$

更新二阶矩： $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g^2(\theta)$

纠正一阶矩： $\hat{m}_t = \frac{\tilde{m}_t}{1 - \beta_1^t}$

纠正二阶矩： $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$

计算参数更新量： $\Delta \theta = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$

更新参数： $\theta_{t+1} = \theta_t + \Delta \theta$

end while

6.10 关于优化算法的使用

6.7 节曾提到, 优化算法分为两类, 其中一类是以动量为核心的算法; 另一类是以自适应为核心的算法。当然, 这两类算法之间也存在着一定的重叠。那么它们各自都有什么特点呢?

以动量为核心的算法更容易在山谷型的优化曲面中找到最优解, 如果优化曲面在某个方向震荡严重, 而在另外一些方向趋势明显, 那么基于动量的算法能够把握这种趋势, 让有趋势的方向积累能量, 同时让震荡的方向相互抵消。但是, 如果趋势不明显呢? 可以想象一下西班牙斗牛, 每一次牛冲向红布时都是积攒了一定的能量的, 而当它冲向红布时, 红布突然撤开, 牛必然会向前冲出一段才能停下来。基于动量的算法也可能遇到这样的问题, 如果趋势不够明显, 那么优化参数的路径必然会存在一些绕弯的情况。

以自适应为核心的算法容易在各种场景下找到平衡, 对于梯度较大的一些场景, 它会适当地减少更新量; 而对于梯度较小的一些场景, 它又会适当地增加更新量, 所以实际上是对优化做了一定的折中。当然, 对于一些复杂且难以优化的场景来说, 这样的方法确实提高了优化效果, 但是对于一些场景不是很复杂的优化问题来说, 这样的限制实际上阻碍了优化的快速进行。虽然这一类算法很优秀, 但是在很多论文中依然使用经典的梯度下降法, 恐怕和这个原因有关。

当然, 理论上结合两者的算法效果应该更好, 因此 Adam 和它的一些改进算法的效果通常不错, 但是其计算量也会相应地增加一些, 这一点在使用时同样要权衡考虑。

参考文献

- [1] Qian N. On the momentum term in gradient descent learning algorithms.[J]. Neural Netw, 1999, 12(1):145-151.
- [2] A method for unconstrained convex minimization problem with the rate of convergence[C]// O(1/k²). Soviet Mathematics Doklady. 1983.
- [3] Duchi J, Hazan E, Singer Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization[M]. JMLR.org, 2011.
- [4] Ruder S. An overview of gradient descent optimization algorithms[J]. arXiv preprint arXiv: 1609.04747, 2016.
- [5] T.Tieleman, and G. Hinton. RMSProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning. Technical report, 2012.

- [6] Zeiler M D. ADADELTA: An Adaptive Learning Rate Method[J]. Computer Science, 2012.
- [7] Kingma D P, Ba J. Adam: A Method for Stochastic Optimization[J]. Computer Science, 2014.
- [8] Timothy Dozat. Incorporating Nesterov Momentum into Adam. ICLR Workshop, (1):2013-2016, 2016.

7

深度学习训练技巧

深度学习有时被大家调侃为中医，即实验科学，因为原理本身并不特别复杂，而效果好坏更多依赖于各种训练技巧。对此，本文作者的观点是，如果要做大的创新，比如设计一套比 TensorFlow 之类的更完美的深度学习平台或者提出一套新的有效模型，都需要很深的理论功底和实践技术，并不是掌握简单的技巧就可以达到的；如果是简单的实验或者业界比较普通的应用，也许理解基本原理后的调参就能做到差强人意。

本章主要介绍数据预处理、权重初始化和正则化技术。

7.1 数据预处理

数据预处理在传统机器学习中非常重要，在深度学习的应用中同样重要，事实上，将数据进行归一化（Normalization）或者白化（Whitening）处理后，算法效果往往可以得到明显提升。

实际中，预处理往往和所采用的具体模型以及面对的具体数据相关，采用哪种预处理方法需要结合实际进行考虑。

以下列举一些常用的归一化方法。

1. 减均值

这是最简单的数据归一化方法，就是所有样本都减去总体数据的平均值。这种初始化方法适合那些各维度分布相同的数据，比如数据的各维度都服从高斯分布，减去各维度均值后就都变为 0 均值了。

2. 大小缩放

统一在 $[-1, 1]$ 或者 $[0, 1]$ 区间内的数据更利于模型的处理, 如果不同维度的取值差异较大, 则可以通过大小缩放的预处理方法达到统一尺度。比如灰度图像的像素取值范围为 $[0, 255]$, 通过各像素除以 255 就可以缩放到 $[0, 1]$ 区间。

3. 标准化

数据标准化一般是指各维度减均值除方差, 这是最常用的归一化方法。各维度之间的协方差矩阵由于是半正定的, 因此可以利用矩阵分解的方法 (例如 SVD) 将协方差矩阵分解得到特征向量和对应的特征值, 再将特征值从大到小排列, 忽略特征值较小的维度, 从而达到降维的效果。如果利用特征值进一步对特征空间的数据进行缩放, 就是进行白化操作。

7.2 权重初始化

由于深度学习的优化是非凸优化问题, 不同的初始化往往导致完全不同的收敛速度和效果, 所以在开始模型训练之前, 寻找最合适的权重初始化方法也是非常重要的。比较常见的权重初始化方法有:

1. 全零初始化

全零初始化即所有变量均被初始化为 0, 这应该是最笨、最省事的随机化方法了。然而这种偷懒的初始化方法非常不适合深度学习, 因为这种初始化方法没有打破神经元之间的对称性, 将导致收敛速度很慢甚至训练失败。

2. 随机初始化

随机初始化即权重初始化为 0 附近的随机值, 例如高斯采样或均匀采样。这种方法轻松打破了神经元之间的对称性, 但比较难把握权重的大小与相关神经元数量的关系。比如输入神经元由 100 增加到 10000 时, 如果初始化的值大小范围不变, 则对应的输出值方差就会出现较大的差异, 经验表明, 这将导致收敛速度较慢甚至失败。

3. 方差校准

为了避免随机初始化方差不稳定的问题, 可以利用数据量的大小来调节初始化的数值范围, 方法是每个初始化值都除以 $\sqrt{n^{[1]}}$, 其中 n 是输入的维数。类似的校准方法还可以将初

始化范围修改为：

$$w \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \right]$$

其中 U 表示均匀分布^[1]。

何恺明等人提出了一种专门针对 ReLU 神经元的初始化方法^[2]，他们认为在 ReLU 神经元系统中初始化参数都应该除以 $\sqrt{n/2}$ 。

bias 因为占比不大，初始化一般直接采用全 0 或很小的数字，或者和权重向量同等对待。

最近，Batch Normalization（批量规范化）的出现在一定程度上缓解了初始化问题。前向传播时，针对激活函数输出，利用 mini-batch 的数据均值和方差进行规范化，从而使数据在每一层的均值和分布都相对稳定。关于 Batch Normalization 的具体介绍我们留到第 22 章。

7.3 正则化

7.3.1 提前终止

提前终止（Early Stopping）是机器学习领域非常通用的简单正则化方法，在决策树等模型中得到广泛应用。提前终止在深度学习领域中的应用也大同小异，在训练过程中随时关注模型的效果，当验证集（Validation Set）上的误差不再减小甚至增大时停止训练。

在深度学习中采用提前终止防止过拟合的具体做法是：在每一轮训练（Epoch）结束时，计算验证集上的损失函数，如果损失函数不再下降或者下降较少时停止训练。当然，为了避免只看一轮迭代带来较大误差的问题，也可以多看几个 Epoch，如果连续几个 Epoch 损失函数都比之前高的话就可以停止训练了。

7.3.2 数据增强

在深度学习应用中训练数据往往不够，可以通过添加噪声、裁剪等方法获取更多的数据。另外，考虑到噪声多种多样，可以通过添加不同的噪声获取更多类型的数据。比如，图片可以在不同的位置裁剪出小一些的图片，也可以通过旋转、扭曲、拉伸等不同方法生成不同的数据。

例如，本书作者在手写识别项目中就用到了数据增强，而且效果非常显著。对于图 7-1 中的“Augmentation”手写单词，分别进行了模糊化（见图 7-2）、对比度（见图 7-3、图 7-4）、

拉伸（见图 7-5、图 7-6）、旋转（见图 7-7）等数据增强。

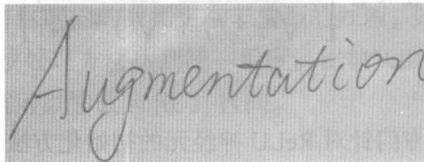


图 7-1 “Augmentation” 手写单词原图

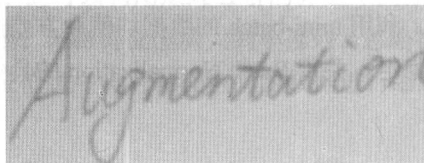


图 7-2 模糊化

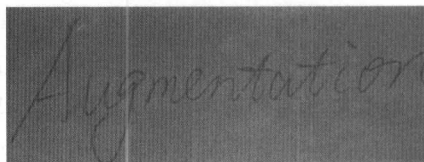


图 7-3 对比度减弱



图 7-4 对比度增强

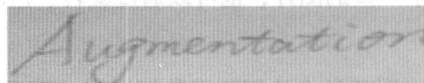


图 7-5 减少高度

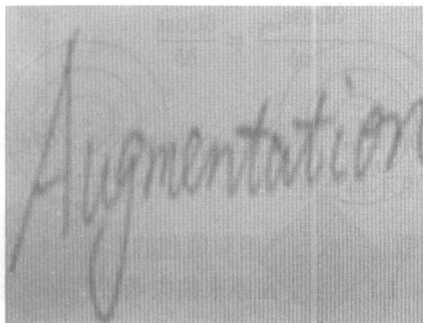


图 7-6 增加高度

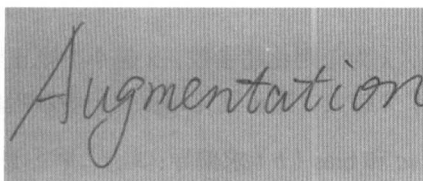


图 7-7 向右旋转一定角度

7.3.3 L2/L1 参数正则化

深度学习的 L2/L1 正则化完全沿袭了传统机器学习。

从形式上看, L2 指的是二范数, 一般写作平方和的形式。下面以分类的损失函数(负的最大似然)为例进行说明。L2 相当于在原来损失函数的基础上多了一项 $\frac{\lambda}{2n} \sum_w w^2$:

$$\text{Loss}_{\text{reg}} = \text{Loss} + \frac{\lambda}{2n} \sum_w w^2 = - \sum_{i=1}^N y_i \log(\hat{y}_i) + \frac{\lambda}{2n} \sum_w w^2$$

其中 n 为训练样本总数, 分母多一个系数 2 只是为了方便求导, 因为平方求导后会多一个常数系数 2。 λ 为正则化超参数, λ 越小, 则正则化所起的作用越小, 模型主要在优化原来的损失函数; λ 越大, 则正则化越重要, 参数趋向于 0 附近。

上面的损失函数对参数 w 和 bias (b) 求偏导:

$$\frac{\partial \text{Loss}_{\text{reg}}}{\partial w} = \frac{\partial \text{Loss}}{\partial w} + \frac{\lambda}{n} w$$

$$\frac{\partial \text{Loss}_{\text{reg}}}{\partial b} = \frac{\partial \text{Loss}}{\partial b}$$

其中对 w 的偏导多了一项 $\frac{\lambda}{n}w$ ，而对 bias 的偏导不变。依此类推，随机梯度下降时 w 的更新也会进行类似变化，这里不再详述。

类似地，L1 指的是一范数，一般写作绝对值和的形式。下面同样以分类的损失函数（负的最大似然）为例进行说明。L1 相当于在原来损失函数的基础上多了一项 $\frac{\lambda}{n} \sum_w |w|$ ：

$$\text{Loss}_{\text{reg}} = \text{Loss} + \frac{\lambda}{n} \sum_w |w| = - \sum_{i=1}^N y_i \log(\hat{y}_i) + \frac{\lambda}{n} \sum_w |w|$$

其中 n 为训练样本总数， λ 为正则化超参数， λ 越小，则正则化所起的作用越小，模型主要在优化原来的损失函数； λ 越大，则正则化越重要，参数趋向于 0 附近。

上面的损失函数对参数 w 和 bias (b) 求偏导：

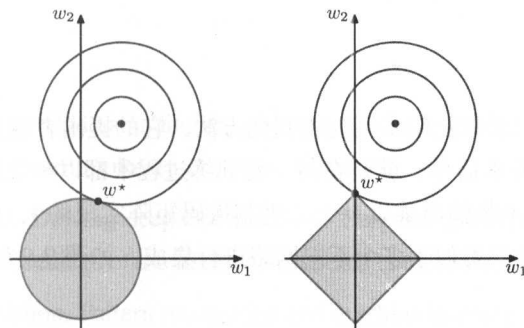
$$\begin{aligned} \frac{\partial \text{Loss}_{\text{reg}}}{\partial w} &= \frac{\partial \text{Loss}}{\partial w} + \frac{\lambda}{n} \text{sign}(w) \\ \frac{\partial \text{Loss}_{\text{reg}}}{\partial b} &= \frac{\partial \text{Loss}}{\partial b} \end{aligned}$$

类似地，对 w 的偏导多了一项 $\frac{\lambda}{n} \text{sign}(w)$ ，其中 $\text{sign}(w)$ 表示 w 的符号， w 为正时取 1， w 为负时取 -1， w 为 0 时一般取 0。而对 bias 的偏导不变。依此类推，随机梯度下降时 w 的更新也会进行类似变化，这里不再详述。

关于 L2/L1 正则原理，一般存在两种解释方法。

- 机器学习经典书籍 *Pattern Recognition and Machine Learning* (PRML) [3] 中的解释为，L2/L1 是添加一个参数 w 取 0 附近的先验，如图 7-8 所示，灰色部分表示相应的先验，左侧为 L2 正则，右侧为 L1 正则，同心圆的圆心表示数据上的最优解，等高线表示到最优解距离相同的点。可以看出，先验希望参数保持在零点附近，当然，L2 的平方和形式决定了其函数图像在二维情况下是圆形，在三维情况下为球面，更多维可以理解为超球面；而 L1 的绝对值形式则决定了在二维情况下为菱形，在三维情况下为正八面体，更多维可以理解为正多面体。

- 贝叶斯学派的观点则认为，L2 正则相当于为参数 w 增加了协方差为 $\frac{1}{\lambda}$ 的零均值高斯分布先验；L1 正则相当于为参数 w 增加了拉普拉斯先验。

图 7-8 L2 正则与 L1 正则的图示^[3]

7.3.4 集成

集成 (Ensemble) 即多个模型进行融合。

生成多个模型的方法有多种, 比如:

- 对数据进行放回重采样的 Bagging (Bootstrap Aggregating) ——从数量为 n 的原始数据 D 中分别独立随机抽取 n 次, 由于是放回重采样, 每次抽取的候选集都是同样的 n 个数据, 这样得到的新数据集用于训练模型。重复这个过程, 就会得到多个模型。
- Boosting——先针对原始数据训练一个比随机分类器性能要好一点的模型, 然后用该分类器对训练数据进行预测, 对预测错误的数据进行加权, 从而组成一个新的训练集, 重新训练即可得到新的模型。
- 不同的训练数据——比如要对视频进行分类, 部分模型用语音数据, 部分模型用字幕图像数据。
- 不同的模型结构——比如有的卷积模型用三层卷积, 有的用五层卷积, 同样的训练数据也可以出来不同的模型。下节将介绍的 Dropout 和 Dropconnect 也属于这种类型。

合并多个模型的方法也有多种, 比如:

- 选择验证集上效果最好的模型。
- 对多个模型进行投票或取平均值。
- 对多个模型的预测结果进行加权平均。

7.3.5 Dropout

Dropout 是深度学习领域比较常用的正则化方法，它的提出者就是深度学习祖师爷 Hinton^[4]。Dropout 的思想非常简单，就是在每一轮训练过程中都以一定概率去掉一些节点（实际操作可能是通过激活函数输出乘以一个二进制掩码矩阵实现的），由于每一轮去掉的节点并不一样，Dropout 的效果类似于多个不同网络进行集成，如图 7-9 所示。

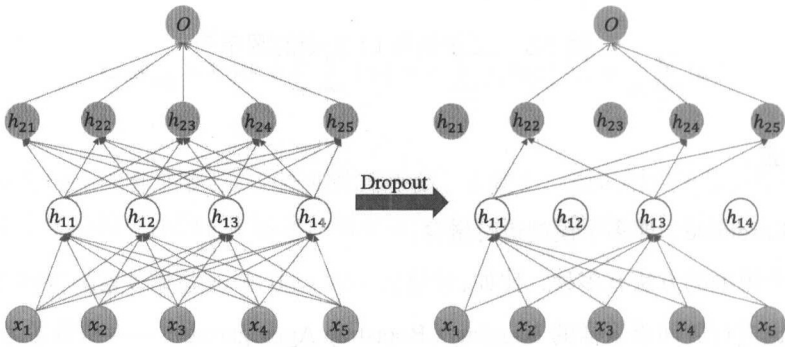


图 7-9 Dropout 示意图

Dropout 直接放弃节点还是有点简单、粗暴，更细粒度的是可以选择只去掉一些边而不是整个节点，这就是 Dropconnect，一种随机放弃连接边的正则化方法，如图 7-10 所示。

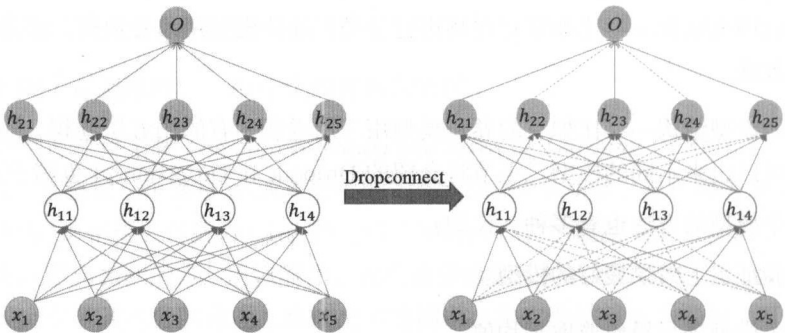


图 7-10 Dropconnect 示意图

除了前面所提及的正则化方法，迁移学习（含多任务学习）、利用无监督模型初始化、二值化网络、模型压缩等都可以看作防止过拟合的方法。

参考文献

- [1] X Glorot, Y Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*. 2010, 9:249-256.
- [2] He, Kaiming, et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision*. 2015.
- [3] Christopher M.. Bishop. *Pattern recognition and machine learning*. Springer, 2006.
- [4] Hinton, Geoffrey E., et al. Improving neural networks by preventing co-adaptation of feature detectors, *arXiv preprint arXiv:1207.0580* (2012).
- [5] Wan L, Zeiler M, Zhang S, et al. Regularization of neural networks using dropconnect. *Proceedings of the 30th international conference on machine learning (ICML-13)*. 2013: 1058-1066.

8

深度学习框架

深度学习能够快速在全世界流行起来，在很大程度上得益于文档齐全、代码清晰、算法完备的深度学习框架。本章将介绍一些目前比较流行的深度学习框架，包括 Theano^[1]、Torch^[2]、PyTorch^[3]、Caffe^[4]、TensorFlow^[5]、MXNet^[6]、Keras^[7] 等框架。在深度学习早期，Theano 是非常受欢迎的，可以说是深度学习框架的鼻祖，很多人都是基于它真正跨入深度学习的大门的；后来 Caffe 由于其高效性迅速抢占了很大一部分市场；这两年名声大噪的 AlphaGo 刚开始是基于 Torch 开发的，后来又转向 TensorFlow。

8.1 Theano

8.1.1 Theano

Theano^[1] 的主要开发者是来自 Yoshua Bengio 领衔的蒙特利尔（University of Montréal）LISA 组（现为 MILA 组）的学生，该框架于 2008 年左右开源，并在 2012 年借 AlexNet 东风迅速流行开来。

Theano 是一个可以运行在 CPU 或 GPU 上的 Python 库，相关的特征包括：

- 紧密集成 NumPy^[8]；
- 擅长处理多维数组的计算；

- Theano 更像一个研究平台而非深度学习库，需要从底层开始写代码构建模型，并不像后来出现的 Caffe、TensorFlow 之类的那样灵活；
- 近年来很多基于 Theano 的开源框架流行起来，包括 Keras 等。

8.1.2 安装

Theano 要求安装 Python、NumPy、SciPy、BLAS 等依赖库，其官方网站^[1]上提供了 Ubuntu、Mac OS、Windows、CentOS 等平台的安装说明。甚至可以直接使用 pip 安装 Theano：pip install theano。

8.1.3 计算图

Theano 引入了计算图的概念，图中的节点主要包括以下三种。

- 变量节点 (Variable Node)：普通的变量，每个变量节点都对应一个 owner 表示其来源。
- 操作节点 (Op Node)：基本的加法、乘法、取和、tanh 等运算都对应 Op 节点。
- 应用节点 (Apply Node)：Op 对某些变量的具体计算或调用就是 Apply，可以说，变量是参数，op 是函数，apply 则是具体的函数执行。

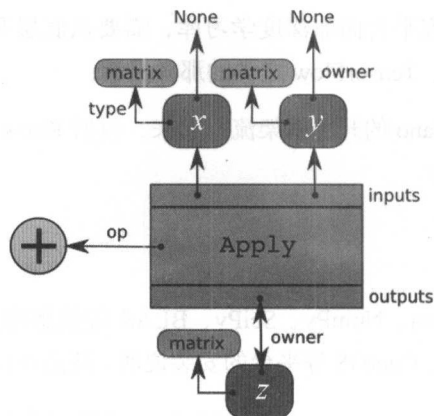
对于一个简单的矩阵运算，Theano 的代码^[1]为：

```
import theano.tensor as T

x = T.dmatrix('x')
y = T.dmatrix('y')
z = x + y
```

对应的计算图如图 8-1 所示，其中箭头代表 Python 对象的引用，'x'、'y'、'z'对应的是变量，'+'对应的是 Op，最大的框对应 Apply，小框 matrix 则表示变量的类型。'x'和'y'的 owner 指向 None，是因为它们不是其他计算的结果；而'z'不一样，指向 Apply 框，表示它是 Apply 对应的计算结果。

引入计算图的最大好处是能够自动求导，tensor.grad() 只需要从输出逆向遍历（穿过 Apply 节点）到输入节点，然后采用链式法则将所有路径对应的梯度相加即可得到总的梯度。当然，对于每个 Op，需要定义其偏导数的计算方式。

图 8-1 矩阵运算对应的 Theano 计算图^[1]

所以，当编写好 Theano 程序时，也就定义了对应的计算图，通过计算图告诉计算机如何由输入算出对应的输出（即前向计算），以及如何由输出得到梯度（即后向计算）。由于计算图的设计和普通的代码编程类似，都可能存在一些冗余的计算，因此引入编译器优化。Theano 的编译器优化除去除冗余计算之外，还可以自动将某些子图的计算转为 GPU 代码执行，从而提高运行效率。

8.2 Torch

8.2.1 概述

Torch^[2] 是一个由 Facebook 主导，实现了各类常见机器学习算法的开源计算框架。由于采用了 C/CUDA+Lua 的架构，Torch 真正实现了“又快又好”。底层依靠 C/CUDA 实现 CPU/GPU 的运算逻辑，其中大量优化过的并行运算保证了程序具有极高的运行效率。上层通过高效而轻量的黏合剂语言 Lua 实现了模型和算法的结构化流程定义，使程序的表达清晰且接近于自然语言，从而拥有更好的开发效率和更强的可读性。可以说，Torch 是 Caffe 的速度和 Tensorflow 的简洁的完美结合体。

总的来说，Torch 更像是一个提供了各方面计算框架的科学计算生态系统，包括但不限于统计学习、计算机视觉、并行计算、语音识别、自然语言处理等。此外，Torch 还拥有丰富的社区支持，为开发者提供了大量的辅助工具包。大多数应用都有对应的参考代码和工具包可供借鉴。

在深度学习方面, Torch 更是具有得天独厚的优势。构建 Torch 网络就像搭积木一样。使用者可以不用花费大量时间深入了解积木本身的实现细节, 而只需关注自己的算法和应用场景, 并按照各个积木的功能, 把自己需要的模型一步步搭建起来再进行训练即可。常见的深度学习架构和后台算法在 Torch 中都已有的实现版本。比如 `cunn` 包实现了对 GPU 的支持, 保证了运算速度和效率的极大优化; `nn` 包实现了神经网络的常见模块和算法, 包括卷积层和各种常见的 `loss` 计算层; `optim` 包实现了 Adam 等常见优化算法; `dnn` 包主要提供了增强学习算法所需要的各个模块; `rnn` 包则实现了高可定制的递归神经网络模型组件和递归单元 LSTM; `dp` 包负责优雅地将流程化代码面向对象化, 同时还封装了大量操作 (如采样方法), 让加载数据、训练和测试模型在数行代码内就可以完成。

Torch 中还提供了大量的例子可供学习。Torch 官方网站定期有最前沿的 Blog 更新, 这些例子都是对近年来深度学习经典论文的实现, 如 Attention 模型和深度增强学习模型 (如 Deep Q Network)。下面给出几个有趣的例子的 github 链接。

利用 Attention 机制训练 MNIST: <http://torch.ch/blog/2015/09/21/rmva.html>

利用 DQN 训练机器玩电子游戏: <https://github.com/Kaixhin/Atari>

利用卷积提取特征, 把图片风格和内容进行融合: https://github.com/jcjohnson/neural-style/blob/master/neural_style.lua

8.2.2 安装

(1) 安装 Torch

在 Mac OS 或 Ubuntu 上安装 Torch 的解释器和核心库:

```
git clone https://github.com/torch/distro.git ~/torch --recursive
cd ~/torch; bash install-deps;
./install.sh
source ~/.bashrc
```

(2) 安装社区扩展包

```
luarocks install package_name
```

也可以从 github 上下载包的源码, 找到后缀为 `.rockspec` 的文件 (一般是 `package_name-scm-1.rockspec`), 使用以下命令安装:

```
luarocks make [后缀为 .rockspec 的文件全名]
```

8.2.3 核心结构

1. Tensor

Tensor 是 Torch 中的基本运算和存储单元，是 Torch 中最核心的类。任何数据，包括网络中的参数、梯度、输入数据、输出数据、标签数据，最终都是由 Tensor 表示的。它的基本形式是一个高维矩阵（读者可类比 Caffe 中的 Blob 和 TensorFlow 中的 NumPy），并在存储数据的基础上提供了许多矩阵和逻辑运算方法。

Tensor 可以完成的功能有很多，除基本的矩阵运算之外，它还提供了如维度变换、单独位运算（激活函数）、BLAS 运算、norm 运算、卷积运算、采样等各种不同类型的函数。因篇幅有限，现在只就以上每种类型抛砖引玉地给出一个例子，如表 8-1 所示。

表 8-1 Tensor 部分函数示例

目的	函数签名	描述
变维	[res]torch.reshape([res,] x, m [,n...])	将 x 转换为 m*n 的矩阵
激活	[res]torch.sigmoid([res,] x)	每个元素求 sigmoid
矩阵乘加	[res]torch.addmm([res,][v1,]M,[v2,]mat1,mat2)	求 $v1 * M + v2 * mat1 * mat2$
P 正则	torch.renorm([res], x, p, dim, maxnorm)	以 maxnorm 进行正则
卷积	[res]torch.conv2([res,] x, k, [, 'F' or 'V'])	k 代表 kernel size
多项分布采样	[res]torch.multinomial([res,]p,n, [,replacement])	概率向量 p 表征分布

Tensor 中 Element-wise 的激活方式实际就是神经网络各个激活层内部所使用的实现方式。p 正则 renorm 主要用于参数的正则（Regression），具体实现是将传入的 maxnorm 值作为一个基准（Baseline），当参数的 p 正则（一般取 2）的值（norm）大于这个阈值时将参数按比例 maxnorm/norm 进行缩小，保证网络不因参数绝对值过大而产生过拟合等问题。采样操作在强化学习网络训练中被大量使用，同时也可以用在数据集的自定义取样方式上。

此外，因 Lua 对函数式编程的良好支持，使用者还可以将自定义的函数应用到 Tensor 中。以下例子简洁地初始化了一个递增序列矩阵。

在命令行输入 th:

```
x = torch.Tensor(2,3)
i = 0
x:apply(function()
    i = i + 1
```

```

return i
end)
> x
1 2 3
4 5 6
7 8 9

```

2. Module

Module 是神经网络中所有组件的基类，往往对应神经网络中的一层，读者可以将其类比 Caffe 中的 Layer。然而与 Caffe 不同的是，Lua 中的对象是一种动态对象，即可以随时地对一个已有对象添加新行为（方法）。所以，Module 可能根据引入包的不同，所能完成的功能和表现的行为也有所不同。按照包的功能分类，Module 在各个包中获得的核心方法如表 8-2 所示。

表 8-2 Module 的核心方法

方法名	所在包	主要作用
updateOutput	nn	正向传播，求预测结果
updateGradInput	nn	反向传播第一步，求对输入的梯度
accGradParameters	nn	反向传播第二步，求参数的梯度
updateParameters	nn	根据学习率和梯度更新参数
reinforce	dpnn	保存增强学习部分的累积奖赏值 R
stepClone	rnn	参数共享（主要用于 step 之间）

下面先介绍 Module 的两个基础方法。

--在Torch中，每一个新的对象必须以此种形式向Torch注册。相当于创建对象和注册对象

```
local Module = torch.class('nn.Module')
```

--看看模块的参数到底是什么形式

```
function Module:parameters()
```

```
    if self.weight and self.bias then
```

```
        return {self.weight,self.bias},{self.gradWeight,self.gradBias}
```

--可以看出，Module的参数实际就是两个表格（Table），里面包含权值、偏置和权值的偏导，这些偏导均为Tensor类型。如果Module是一个组合模块，则该函数返回其所有子

Module的参数，具体可以参见后面的Sequential类

--实现梯度下降，这只是一种通用实现，具体到不同的模块参数更新方式也可能不同


```

function Module:updateParameters(learningRate)
    local params, gradParams = self:parameters()
    --如果该Module有参数
    if params then
        --对所有参数执行params = params - gradParams * learningRate
        for i=1,#params do
            params[i]:add(-learningRate, gradParams[i])
        end
    end
end
end

```

3. Linear

线性模块就是全连接层，该层实现 $y = Ax + b$ 的逻辑。模块的定义如下：

```

--inputDimension表示输入的维数，outputDimension表示输出的维数，bias默认为true，
表示需要偏置
module = nn.Linear(inputDimension, outputDimension, [bias = true])

```

与 Caffe 不同，Torch 中的模块可以自动匹配单实例输入和批量输入两种形式，也就是说，并不需要显式地在训练前指定 batchsize 的数值。各个模块会根据设定的 inputDimension 和实际传入的 input 的 size 来推断该层的 batch 数量。

Linear 作为一个简单的线性计算模块（层），其核心函数有 updateOutput、updateGradInput 和 accGradParameters 三个，分别对应正向传播、反向传播和计算参数梯度。下面简单介绍 updateOutput 函数。

```

--正向传播实现的公式：output = input * weights + bias
function Linear:updateOutput(input)
    --如果input是一个一维向量
    if input:dim() == 1 then
        self.output:resize(self.weight:size(1))
        --如果有偏置，先将偏置复制到output的Tensor上，即output=bias
        --如果没有偏置，则将其置0
        if self.bias then self.output:copy(self.bias)
        else self.output:zero() end
        --计算公式：output=output + input * weight
        self.output:addmv(1, self.weight, input)
    end
end

```

--input的维度为2, 表示输入的是一个batch, 表示为(batchsize, IDimension)

```
elseif input:dim() == 2 then
```

--如果是batch, 则变为矩阵相乘

```
self.output:addmm(0, self.output, 1, input, self.weight:t())
```

```
if self.bias then
```

```
    self.output:addr(1, self.addBuffer, self.bias)
```

```
end
```

```
else
```

```
    error('input must be vector or matrix')
```

4. Sequential

Sequential 组件继承自 container 类, 是 Torch 中最重要的容器类, 主要功能是将其包含的各个子类按照串行顺序执行 (顺序由子模块的先后加入次序决定)。因为常见的神经网络都是数据单向流动的 DAG 图, 这个类作为拼装底层基础模块的中间件, 几乎在所有的网络中都有涉及。以下是 Sequential 的使用方法。

```
model = nn.Sequential()
```

```
model:add(nn.Linear(10, 20))
```

```
model:add(nn.Linear(20, 30))
```

```
model:insert(nn.Linear(20, 20), 2)
```

```
> model
```

```
nn.Sequential {
```

```
  [input -> (1) -> (2) -> (3) -> output]
```

```
  (1): nn.Linear(10 -> 20)
```

```
  (2): nn.Linear(20 -> 20)    -- 插入到第二个位置的全连接层
```

```
  (3): nn.Linear(20 -> 30)
```

```
}
```

8.2.4 小试牛刀

1. 训练一个与或门感知机

```
require 'nn'
```

```
local model = nn.Sequential()
```

```

local inputSize, outputSize, hiddenSize = 2,1,20
model:add(nn.Linear(inputSize,hiddenSize))
model:add(nn.Tanh())      --将输出映射到(-1,1)区间
model:add(nn.Linear(hiddenSize, outputSize))

--使用均方差作为评估标准 (Mean Squared Error criterion)
criterion = nn.MSECriterion()
for i = 1,2500 do
    local input = torch.randn(2);      --随机取两个数
    local output = torch.Tensor(1);    --Label也是Tensor
    if input[1]*input[2] > 0 then      --如果输入的两个数同号，则输出1
        output[1] = -1                --同号标签为负1
    else
        output[1] = 1                  --异号标签为正1
    end
    --生成loss
    criterion:forward(model:forward(input), output)

    --训练过程实际分为以下三步
    -- (1) 将之前计算的梯度清0
    model:zeroGradParameters()
    -- (2) 反向传播计算梯度
    model:backward(input, criterion:backward(model.output, output))
    -- (3) 以0.01的学习率更新参数
    model:updateParameters(0.01)
end

```

2. 训练基于 rnn 的语言模型

因篇幅有限，这里只重点介绍如何构造 rnn 语言模型，以及如何训练模型的主干代码。数据准备、预处理、后处理等细节均省略在了 loadBatch() 函数中。关于如何根据不同的数据来源准备训练数据，读者可以参考 dp 包的 data 目录下如何对各个常见数据集进行处理的例子。

```

--配置参数
batchSize = 10
nsteps = 5      --rnn迭代次数，也就是语言模型的窗口大小

```

```

hiddenSize = 64
nIndex = 10
lr = 0.1
--构造简单的rnn语言模型，其中使用了LookupTable模块，它实现了语言模型的embedding
过程，即将单词（或字母）的索引转化为隐层向量
--用h_(t)表示t时刻该模块的输出，input_(t)表示输入，Recurrent模块的内在逻辑是
-- h_(t) = transfer(Linear(h_(t-1)) + LookupTable(input_(t)))
local r = nn.Recurrent(
    hiddenSize, nn.LookupTable(nIndex, hiddenSize),
    nn.Linear(hiddenSize, hiddenSize), nn.Sigmoid(),
    nsteps)
--将每一步输出的h_(t)送到Softmax层，得到最终输出
local rnn = nn.Sequential()
    :add(r)
    :add(nn.Linear(hiddenSize, nIndex))
    :add(nn.LogSoftMax())

--开始训练
local iteration = 1
while true do
    --（1）从数据集中获取一批实例（batch），对rnn来说，inputs和targets均为列表，
    里面包含每个时间点（step）的输入和目标
    local inputs, targets = loadBatch()
    --inputs = {input1, input2, ...}, targets = {target1, target2, ...}

    --（2）让batch中的序列在rnn中正向传播
    rnn:zeroGradParameters() --清空上一步留下的梯度值
    rnn:forget()              --清空rnn模型中的各种暂存变量
    --保存输出列表和最终的loss，每一轮迭代都需要将上一轮的值清空
    local outputs, loss = {}, 0
    for step=1,nsteps do
        outputs[step] = rnn:forward(inputs[step])
        --rnn的loss是每一步loss之和
        loss = loss + criterion:forward(outputs[step], targets[step])
    end
    print(string.format("Iteration %d ; NLL err = %f ", iteration, loss))

```

```

-- (3) 按时间序列反向传播 (BPTT)
--保存对输出的梯度和对输入的梯度
local gradOutputs, gradInputs = {}, {}
for step=nsteps,1,-1 do  --按正向传播的逆序进行反向传播
    gradOutputs[step] = criterion:backward(outputs[step], targets[step])
    gradInputs[step] = rnn:backward(inputs[step], gradOutputs[step])
end

-- (4) 按照学习率更新参数
rnn:updateParameters(lr)
iteration = iteration + 1
end

```

8.3 PyTorch

8.3.1 概述

PyTorch^[3] 可以看作是 Torch 的 Python 版, 由于 Python 语言的风行, 使得 Torch 的开发者将这一优秀的框架移植到了 Python 平台上。PyTorch 除可以让开发者享受 Python 的便捷外, 还使用了“define by run”的动态图构建思路, 使得模型的构建变得更加灵活。

8.3.2 安装

访问 <http://pytorch.org/> 网站, 在“Get Started”部分根据实际情况选择适合自己的安装环境, 本书作者的安装环境如图 8-2 所示。

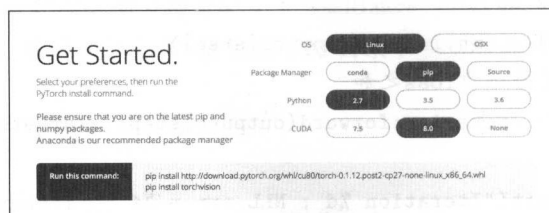


图 8-2 PyTorch 的安装选项页面

8.3.3 核心结构

PyTorch 的核心结构和 Torch 的结构类似，这里就不再赘述了。

8.3.4 小试牛刀

以下代码来自 PyTorch 的官方样例^[3]，注释是本书作者添加的。

```
from __future__ import print_function
import argparse
import torch # PyTorch的基础库
import torch.nn as nn # PyTorch的神经网络库
import torch.nn.functional as F # PyTorch的常用函数库
import torch.optim as optim # PyTorch的优化库
from torchvision import datasets, transforms # PyTorch的视觉操作库
from torch.autograd import Variable # PyTorch的自动求导库

# Training settings
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                    help='input batch size for training (default: 64)')
parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                    help='input batch size for testing (default: 1000)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train (default: 10)')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate (default: 0.01)')
parser.add_argument('--momentum', type=float, default=0.5, metavar='M',
                    help='SGD momentum (default: 0.5)')
parser.add_argument('--no-CUDA', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')
args = parser.parse_args()
```

```

args.CUDA = not args.no_CUDA and torch.cuda.is_available()

torch.manual_seed(args.seed)
if args.CUDA:
    torch.cuda.manual_seed(args.seed)

# pin_memory使得Host到GPU端的内存拷贝更加快速
kwargs = {'num_workers': 1, 'pin_memory': True} if args.CUDA else {}
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                    transform=transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize((0.1307,), (0.3081,))
                    ])),
    batch_size=args.batch_size, shuffle=True, **kwargs)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])),
    batch_size=args.batch_size, shuffle=True, **kwargs)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 以下定义了网络结构
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        # 以下定义了前向计算, 后向计算会自动生成
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))

```

```

        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        # log_softmax方法等价于log(softmax(x))
        return F.log_softmax(x)

model = Net()
# 使用GPU计算
if args.CUDA:
    model.cuda()
# 优化算法初始化
optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if args.CUDA:
            data, target = data.cuda(), target.cuda()
        data, target = Variable(data), Variable(target)
        # 清空旧的梯度
        optimizer.zero_grad()
        # 前向计算
        output = model(data)
        # nll为Negative Likelihood loss, 返回的loss为output[target]
        loss = F.nll_loss(output, target)
        # 反向计算
        loss.backward()
        # 优化更新参数
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tloss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))

def test(epoch):

```



```

model.eval()
test_loss = 0
correct = 0
for data, target in test_loader:
    if args.CUDA:
        data, target = data.CUDA(), target.CUDA()
    # 在测试时将变量的属性设置为volatile=True会节省内存
    data, target = Variable(data, volatile=True), Variable(target)
    output = model(data)
    test_loss += F.nll_loss(output, target).data[0]
    pred = output.data.max(1)[1] # get the index of the max log-probability
    correct += pred.eq(target.data).cpu().sum()

test_loss = test_loss
test_loss /= len(test_loader) # loss function already averages over batch
size
print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%) \n'.
      format(
          test_loss, correct, len(test_loader.dataset),
          100. * correct / len(test_loader.dataset)))

for epoch in range(1, args.epochs + 1):
    train(epoch)
    test(epoch)

```

8.4 Caffe

8.4.1 概述

Caffe^[4]是由伯克利视觉中心实验室开发的深度学习框架。Caffe 采用了 C++/CUDA 的底层架构，具有较强的稳定性和较高的运行效率，这使得它一被推出就获得了广泛的关注和使用。然而，由于近年来支持脚本语言的深度学习框架盛行，其轻量级编程模式带来的快速开发优势也越来越明显，Caffe 不够便捷这一短板（需要预编译，结构僵化）使得它的热度有所下降。不过，Caffe 自身也做出了调整，提供了支持 Python 的编程接口 PyCaffe。但是瑕不掩瑜，与其他深度学习框架相比，Caffe 所具有的优势仍然突出，主要表现为以下 4 点。

1. 配置清晰

Caffe 的网络定义必须遵守 Google 的 Protobuf 规范,大部分常见网络 (VGGNet16、ResNet 等) 都有现成的 Protobuf 模板。从某种程度上说, Caffe 实现了“基于配置的编程”, 开发者甚至不需要编写任何源代码, 只需集中精力关注构造模型算法本身——从已有模板派生模型, 设置超参数, 并按规则编写相应的模型定义文件。

2. 代码易于扩展

由于 C++ 本身的语言规范和 Protobuf 的引入, 不同 Caffe 程序的代码的绝大部分是相同的。从一个需求到另一个需求往往只需要做出少量调整, 修改极少的类, 甚至只需要修改网络配置文件。

3. 高速

C++/CUDA 的架构在 CPU/GPU 内核速度上的天然优势, 使得 Caffe 在运行效率上大大领先于其他基于脚本语言 (如 TensorFlow) 的深度学习框架。不仅如此, Caffe 内部还封装了大量的性能优化算法, 使得并行运算速度进一步提高。高速特性正是研究界和工业界选择 Caffe 的主要原因。

4. 社区支持

Caffe 社区拥有相当丰富的模型和代码资源。Caffe 的 MODEL ZOO 不仅提供了常见的深度学习算法所对应的网络配置文件, 也支持下载已经训练好的模型。开发者在开发特定模型时, 可以根据自己的需求稍微修改 MODEL ZOO 里的已有网络文件, 然后下载预训练好的模型, 在此基础上进行微调。此外, 如果开发者有特殊的需求, 需要对代码进行更改 (如添加自定义的层), Caffe 社区也提供了大量相关例程可供参考。

8.4.2 安装

基本安装:

```
git clone https://github.com/BVLC/Caffe.git
cd $CAFFE_HOME
cp Makefile.config.example Makefile.config
make -j8
```

如果需要 Caffe 提供的 Python 接口,则去掉 Makefile.config 中的 WITH_PYTHON_LAYER := 1 这一行注释;然后在确保安装了各种 Python 库之后,执行 make pyCaffe。如果需要使用 cuDNN,则将 Makefile.config 中的 USE_CUDNN := 1 这一行注释去掉即可,当然,需要确保已正确安装了 cuDNN。

8.4.3 核心组件

1. Blob 组件

Blob 在 Caffe 中所表示的概念与数据库中常用的概念一致,表示在深度学习网络中用到的所有大型数据块,包括输入数据、输出数据、网络参数等。它的内部实现又可进一步划分为三大数据块,每个数据块实际上都是一个高维矩阵。

```
shared_ptr<SyncedMemory> data_;           //存储数据本身
shared_ptr<SyncedMemory> diff_;          //在反向传播中存储数据对应的梯度
shared_ptr<SyncedMemory> shape_data_;    //存储形状数据
```

一个 Blob 存储的信息就等于上述三个同步内存块 (SyncedMemory) 之和。Blob 组件既作为基础的数据单元,其中包含的方法当然也全是针对操作这些数据块的。以下是最常用的对形状数据的变换——Reshape 方法,它负责转换 Blob 所表征的形状。

```
template <typename Dtype>
//所谓Dtype就是单元数据的基本数据类型,通常为float
void Blob<Dtype>::Reshape(const vector<int>& shape) {
    //count_统计在新的形状信息下单位数据的个数
    count_ = 1;
    shape_.resize(shape.size());
    //为shape_data_分配内存
    if (!shape_data_ || shape_data_>size() < shape.size() * sizeof(int)) {
        shape_data_.reset(new SyncedMemory(shape.size() * sizeof(int)));
    }
    //得到存储形状数据的内存块的指针,将新的形状赋值过去
    int* shape_data = static_cast<int*>(shape_data_>mutable_cpu_data());
    for (int i = 0; i < shape.size(); ++i) {
        count_ *= shape[i];
        shape_[i] = shape[i];
        shape_data[i] = shape[i];
    }
}
```

```

}

//如果新的单元数据个数大于原来的数据个数，则需要重新分配内存
if (count_ > capacity_) {
    capacity_ = count_;
    data_.reset(new SyncedMemory(capacity_ * sizeof(Dtype)));
    diff_.reset(new SyncedMemory(capacity_ * sizeof(Dtype)));
}
}

```

从上面最后两行代码可以看出，`data_`和`diff_`两个数据块本身是没有形状概念的，它们只是由智能指针表示的一维内存块，只负责简单地存下来数据，而怎么规划这些数据则由形状数据块`shape_data_`所决定。换句话说，如果`reshape`时新的数据个数不大于原来的数据个数，那么`data_`和`diff_`都不需要做出任何改变，改变的只是对它们的描述方式（形状）。

`Blob`作为一种数据存储单元，它对外提供的数据访问接口又是怎样的呢？下面是它的核心方法，通过这四个方法可以获取访问数据块的指针。

```

const Dtype* cpu_data() const;
const Dtype* gpu_data() const;
Dtype* mutable_cpu_data();
Dtype* mutable_gpu_data();

```

以上四个方法的实现方式基本相同，都是调用了内部同步内存块的相应方法，随便看下其中一个方法的实现：

```

Dtype* Blob<Dtype>::mutable_gpu_data() {
    CHECK(data_); //使用glog宏，在非法条件下可以中止程序
    return static_cast<Dtype*>(data_>mutable_gpu_data());
}

```

为什么访问接口会有四个方法，而不是只有两个简单的`set/get`方法呢？这样设计主要是因为 GPU 和 CPU 拥有各自的内存区，它们之间很容易出现类似数据不一致的同步问题。一般程序因为只需要使用 CPU，所以`set/get`方法就可以应付。而 Caffe 需要在 GPU 中进行大量并行运算，数据被频繁地在 GPU 和 CPU 之间来回拷贝，如何保证数据的一致性就变得至关重要。Caffe 通过在 `SyncedMemory` 内部维护一个 `head` 指针来实现数据的及时同步。`mutable` 意为“变化的”，当使用 `mutable_gpu_data` 方法获取数据指针时就意味着数据即将被改变，`head` 此时会被置为 `HEAD_AT_GPU`，表示最新的数据在 GPU 显存中，而不在 CPU 内

存中。接下来如果再调用 `cpu_data` 方法，程序会先去检查 `head` 指针的状态，发现最新版本的数据在 GPU 中，因此会自动同步 GPU 数据到 CPU 中，然后置 `head` 为 `SYNCED`（表示这块数据在两个内存区已经同步），并返回给调用者已经同步的数据。所以，调用者在使用这几个方法时需要特别注意，如果要改变所取到的数据，则需要使用带 `mutable` 前缀的后两种方法；如果只是纯读取操作，则选择前两种方法；否则，可能会出现改变了数据而没有改变 `head` 指针的情况，而导致数据未能同步，出现数据不一致问题。

除上述方法之外，`Blob` 还内置了很多运算方法。比如 `update` 方法可以实现 SGD，把 `diff` 数据加到 `data` 中；`sumsq_data` 方法则可以计算 `data` 的 L2 正则，在优化参数以防止过拟合时被调用。

2. Layer 组件

Layer 种类虽多，但它们的核心方法其实只有以下三个：

```
void SetUp(vector<Blob<Dtype>*>& bottom, vector<Blob<Dtype>*>& top)
Dtype Forward(vector<Blob<Dtype>*>& bottom, vector<Blob<Dtype>*>& top)
void Backward(vector<Blob<Dtype>*>& top, vector<bool>& propagate_down,
               const vector<Blob<Dtype>*>& bottom)
```

`SetUp` 实际上就是对层进行初始化。主要分为三步：

- (1) 生成锁（层间参数共享时用到）和一些配置信息，检查 `Blob` 个数是否正确。
- (2) 不同层进行自定义初始化函数 `LayerSetUp`。
- (3) 修改 `bottom` 和 `top` 的形状，分配好 `Blob` 的内存等待接收数据。

`Forward` 和 `Backward` 方法会首先判断是使用 CPU 还是 GPU。事实上，在 Caffe 中大多数层的实现都对应一个 `.cpp` 文件和一个 `.cu` 文件，分别对应于 C++ 的 `forward_cpu` 方法和 CUDA 的 `forward_gpu` 方法（`Backward` 同理）。前向传播主要是对输入 `Blob` 进行处理之后生成输出 `Blob`，后向传播主要是对输出 `Blob` 的 `diff` 处理之后生成输入 `Blob` 和参数的 `diff`。

如果按应用场景分，Caffe 中的 Layer 可以分为以下五类。

- **Vision Layer**，视觉应用方面的层，代表有卷积层（`Convolution`）、池化层（`Pooling`）、反卷积层（`Deconvolution`）等。
- **Loss Layer**，损失函数计算层，代表有最大似然（`SoftmaxWithloss`）、欧式距离（`Euclideanloss`）等。
- **Activation Layer**，激活函数层，将线性转化为非线性，代表有常用的激活函数 `ReLU`、`Sigmoid`、`Tanh` 等。

- Data Layer, 数据加载层, 负责将数据从底层形式 (levelDB、HDF5 等) 转化为 Blob。继承自 BaseDataLayer 的 BasePrefetchingDataLayer 类可以通过多线程预读取数据, 通常可以满足普通数据加载的需求。如果数据形式比较特殊, 开发者可以继承 BasePrefetchingDataLayer, 重载其中的 load 和 forward 函数, 以实现自定义加载数据。
- Common Layer, 通用操作层, 代表有全连接层 (InnerProduct)、特征连接层 (Concat) 等。

3. Net 组件

Net 就是很多层 (Layer) 叠起来所构成的网络, 由 Google 的 Protobuf 格式定义。通常的网络结构需要包含数据加载层和损失函数计算层, 中间的流程部分实际上表征一个有向无环的数据流图, 对应于不同模型的流程框架图。

总的来说, Net 的 proto 文件体现的就是某深度学习算法的实现细节, 里面包含大量相互关联的层, 而每个层中又定义了大量关于该层的实现细节。在实际应用中, 完整的网络模型文件往往复杂、庞大, 为简单起见, 下面只介绍一个简单的 CNN 分类模型的网络结构框架。

```
name: "AlexNet" //以下网络节选了AlexNet的一部分结构
//数据加载层
layer {
  name: "data" //这个数据加载层的名字叫data, 加载预训练模型时参数按名字赋值
  type: "Data" //该层的实现类, 这里使用了data_layer.hpp中定义的数据类
  top: "data" //加载数据文件后, 训练数据被输出到名为data的Blob中
  top: "label" //加载数据文件后, 标签数据被输出到名为label的Blob中
  include { //过滤规则: 只在训练阶段包括该层 (即测试、验证阶段的DAG没有该层)
    phase: TRAIN //测试验证时往往需要再写一个与此Layer相同的层, 将phase改为
  } //TEST, 再把下面的source属性改成测试数据即可
  data_param {
    source: "examples/imagenet/train_lmdb" //训练数据的物理位置
    batch_size: 256 //批处理个数
    backend: LMDB //使用LMDB作为数据库文件格式
  }
} //数据加载层定义结束

//卷积层
layer {
  name: "conv1" //层的名字叫conv1
```

```

    type: "Convolution"           //使用Convolution类
    bottom: "data"                //从data Blob中取数据，也就是数据加载层的输出
    top: "conv1"                  //处理完成后将数据输出到名为conv1的Blob
    convolution_param {           //卷积层的配置
        num_output: 96            //输出96路，即有96个卷积核
        kernel_size: 11           //卷积核大小
        stride: 4                  //卷积的步长
        weight_filler {           //卷积的权值初始化
            type: "gaussian"       //从高斯分布中采样初始化权值
            std: 0.01              //分布的方差为0.01
        }
        bias_filler {             //卷积的偏置初始化
            type: "constant"       //偏置初始化为0
            value: 0
        }
    }
}
} //卷积层定义结束
//激活层
layer {
    name: "relu1"
    type: "ReLU"
    bottom: "conv1"                //conv1 Blob作为输入
    top: "conv1"                  //处理完成后继续输出到conv1
}
//池化层
layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 3
        stride: 2
    }
}
}

```

//以上三层可以作为一个单元继续迭代，增加网络深度，图像信息会越来越抽象，现在直接到全连接层

```
layer {
    name: "fc1"
    type: "InnerProduct"
    bottom: "pool1"
    top: "fc1"
    inner_product_param {
        num_output: 10 //分类的label个数
    }
}

//损失函数层
layer {
    name: "loss"
    type: "SoftmaxWithloss"
    bottom: "fc8"
    bottom: "label" //使用了刚开始的label Blob
    top: "loss" //生成loss
}
```

每一个 top 都是一个 Blob，Net 组件的任务就是负责把每个 Layer 和它要进行处理 Blob 关联起来，也就是形成之前所说的数据流图。Net 的前向和后向传播都很简单，就是按序（Layer 会根据数据流形成一个拓扑序列）调用每个层的前向或后向方法。

4. Solver 组件

网络构造好之后，接下来就是怎么训练参数的问题。Solver，直译为求解器，就是定义训练网络的各项超参数（Hyper-parameters），Solver 默认是使用 SGD 进行训练的。

```
net: "models/bvlc_alexnet/train_val.prototxt" //指向刚才网络文件的位置
test_interval: 1000 //1000轮迭代测试一次网络
base_lr: 0.01 //初始学习率
lr_policy: "step" //学习率衰减策略
gamma: 0.1
stepsize: 100000 //每stepsize学习率衰减一次
max_iter: 450000 //训练的总轮数
weight_decay: 0.0005 //权重衰减以防止过拟合
solver_mode: GPU //用GPU训练
```


简单地说, Caffe 的架构就是: 层参数和网络流中的数据块都由 Blob 表示, Layer 是操作加工这些数据块的功能单元, 各种 Layer 按序拼接起来就形成了 Net, 最后使用 Solver 定义策略来训练 Net。

8.4.4 小试牛刀

运行 MNIST 数据集, 进行数字分类实验主要分为构造数据库文件和定义训练相关配置文件两步。

(1) 运行 `$CAFFE_ROOT/data/mnist/get_mnist.s`, 下载数据集, 得到数据集的 gz 压缩文件, 解压缩后执行 `examples/mnist/create_mnist.sh`, 该脚本主要是使用 `convert_mnist_data.bin` 工具将 mnist 转换为 lmdb 文件, 也可使用 `tools/convert_imageset` 工具把图片转换为 lmdb 格式。

(2) 定义网络和求解器的配置文件。在 `examples/mnist` 目录下已有 `lenet_train_test.prototxt` 和 `lenet.prototxt` 分别作为配置好的训练网络和预测网络, 两者的区别仅仅在于网络最后一层是否需要生成 loss。

求解器配置位于 `lenet_solver.prototxt` 中, 可根据情况修改模式为 `cpu/gpu`, 然后设定迭代轮数等训练参数后, 直接执行 `examples/mnist/train_lenet.sh` 即可开始训练。几小时后, 一个可以区分数字集的模型就诞生了, 是不是很简单?

8.5 TensorFlow

8.5.1 概述

TensorFlow^[5] 是由 Google 开源的一个基于计算图的框架, 由于其灵活性和便捷性, 受到了广大开发人员的关注, 目前也是关注度最高的开源框架之一。

8.5.2 安装

TensorFlow 的安装过程分为直接安装和从源码编译两种, 官方网站 <https://www.TensorFlow.org/install/> 已经对不同机型、不同配置的环境给出了不同的安装方法, 读者可以自行研究。对于国内的读者, 可以使用镜像站点的安装包进行安装, 如 <https://mirror.tuna.tsinghua.edu.cn/help/TensorFlow/>。

8.5.3 核心结构

TensorFlow 的使用过程主要分两个部分:构建计算图和运行计算图。计算图由变量 (Variable) 和运算符 (Op) 构成。比如有一个运算公式 $y = a \times b + a + b$, 用 TensorFlow 实现的代码如下:

```
import TensorFlow as tf
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
c = a * b
d = a + b
y = c + d
```

这样就构建出了一个计算图, 其中的“placeholder”表示占位符, 等到计算图执行时, 我们需要为其填入符合的值才能进行运算。一般来说, 这些占位符相当于计算图的输入, 有了输入才能得到输出。这时的计算图形状如图 8-3 所示。

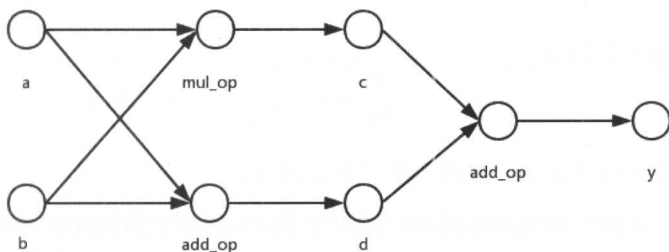


图 8-3 通过代码构建的计算图示意图

构建完计算图后, 我们就可以运行它了。运行时首先需要有一个 Session, 并初始化其中的变量。这个 Session 用于管理与运算相关的内容。

```
sess = tf.Session()
tf.global_variables().run(session=sess)
```

接下来进行运算:

```
sess.run(y, feed_dict={a:3, b:5})
# 23.0
```

在上面的命令中，`Session` 计算了 y 的值，为了写入输入信息，我们传入了一个 `feed_dict` 结构，此时的计算图变成了如图 8-4 所示的样子。

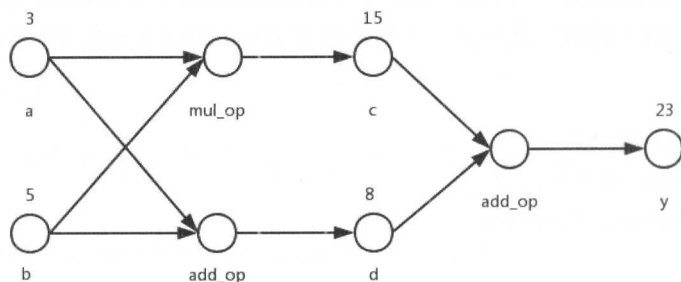


图 8-4 运行后的计算图示意图

除了进行前向计算，计算图还可以进行后向计算。这时我们可以定义 a 对 y 的偏导：

```
grad_a = tf.gradients(y, a)
sess.run(grad_a, feed_dict={a:3, b:5})
# [6.0]
```

根据上面的公式我们知道：

$$\frac{\partial y}{\partial a} = b + 1$$

因此将数字代入可以证明，最终的结果是正确的。

对于机器学习来说，模型训练的核心过程就是前向计算、后向计算和参数更新，实际上这个过程可以被优化器封装起来，这样我们就不需要写这些烦琐的代码了。

8.5.4 小试牛刀

下面就来看看官方^[5]给出的基于 TensorFlow 框架进行 MNIST 数字识别的代码。

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import argparse
import sys
from TensorFlow.examples.tutorials.mnist import input_data
```

```

import TensorFlow as tf
FLAGS = None

# 辅助函数，帮助我们更加方便地进行卷积操作
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

# Pooling函数，帮助我们更加方便地进行Pooling操作
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='SAME')

# 初始化权重参数的函数，参数将以truncated_normal的形式初始化
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

# 初始化偏置项参数的函数，参数将被初始化为0.1
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def deepnn(x):
    # 输入的数据为扁平的784个数字，首先将其变成[batch_num, height, width,
    # channel]的张量形式
    with tf.name_scope('reshape'):
        x_image = tf.reshape(x, [-1, 28, 28, 1])

    # 第一层卷积操作，将1个channel变换到32个channel
    with tf.name_scope('conv1'):
        W_conv1 = weight_variable([5, 5, 1, 32])
        b_conv1 = bias_variable([32])
        h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

    # Pooling操作
    with tf.name_scope('pool1'):

```

```

    h_pool1 = max_pool_2x2(h_conv1)

# 第二层卷积操作，将32个channel变换到64个channel
with tf.name_scope('conv2'):
    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    h_conv2 = tf.nn.conv2d(h_pool1, W_conv2) + b_conv2

# Pooling操作
with tf.name_scope('pool2'):
    h_pool2 = max_pool_2x2(h_conv2)

# 第一层全连接操作，经过上面的运算，此时的参数大小变成了7 * 7 * 64，这里就要将它们映射到1024个特征上
with tf.name_scope('fc1'):
    W_fc1 = weight_variable([7 * 7 * 64, 1024])
    b_fc1 = bias_variable([1024])

    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# Dropout操作，防止特征表示过拟合
with tf.name_scope('dropout'):
    keep_prob = tf.placeholder(tf.float32)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# 第二层全连接操作，将特征变换为10维，每一维对应一个数字的logit值
with tf.name_scope('fc2'):
    W_fc2 = weight_variable([1024, 10])
    b_fc2 = bias_variable([10])

    y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
return y_conv, keep_prob

def main(_):
    # 读取数据

```

```

mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

# 准备好图像和标签两个输入项
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

# 通过构建计算图得到模型结果
y_conv, keep_prob = deepnn(x)

# 计算模型结果和真实结果的差距, 也就是Cross Entropy loss
with tf.name_scope('loss'):
    cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                             logits=y_conv)
    cross_entropy = tf.reduce_mean(cross_entropy)

# 创建优化器, 对Cross Entropy loss进行优化, 也就是进行前向计算、后向计算和
# 梯度更新
with tf.name_scope('adam_optimizer'):
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

# 比较模型结果和真实结果的差距, 计算精确率
with tf.name_scope('accuracy'):
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
    correct_prediction = tf.cast(correct_prediction, tf.float32)
    accuracy = tf.reduce_mean(correct_prediction)

# 创建一个Session
with tf.Session() as sess:
    # 对Session做初始化
    sess.run(tf.global_variables_initializer())
    # 对模型进行20000轮迭代
    for i in range(20000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:
            # 每100轮迭代计算一次精确率
            train_accuracy = accuracy.eval(feed_dict={

```

```

        x: batch[0], y_: batch[1], keep_prob: 1.0})

    print('step %d, training accuracy %g' % (i, train_accuracy))
    # 训练进行的运算
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

    print('test accuracy %g' % accuracy.eval(feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str,
                        default='/tmp/TensorFlow/mnist/input_data',
                        help='Directory for storing input data')
    FLAGS, unparsed = parser.parse_known_args()
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

8.6 MXNet

8.6.1 概述

MXNet^[6] 是一个全功能、灵活可编程和高扩展性的深度学习框架，支持深度学习模型中的最先进技术，包括主要用于图像处理的卷积神经网络（CNN）和主要用于自然语言处理的长短时记忆网络（LSTM）。MXNet 由学术界发起，包括数所顶尖大学的研究人员的贡献，如华盛顿大学和卡内基梅隆大学。目前 MXNet 已经得到亚马逊的支持。

8.6.2 安装

MXNet 官方同样提供了十分详尽的安装流程，读者可以根据实际情况选择适合自己的安装环境，本书作者的安装环境如图 8-5 所示。

8.6.3 核心结构

MXNet 的核心结构包括 NDArray、Symbol、Module 和 Iterator。



图 8-5 MXNet 的安装选项页面

NDArray 是所有数学计算的核心数据结构，它的含义和 Python 中的知名库 NumPy 的 NDArray 含义相近。MXNet 支持在众多平台上快速计算 NDArray，如 CPU、GPU、多 GPU 甚至分布式集群的场景。同时 MXNet 还支持延迟执行等特性。一个 NDArray 有如下几个重要属性，如表 8-3 所示。

表 8-3 NDArray 的重要属性

属性	含义
ndarray.shape	数组的维度大小，对于一个有 n 行 m 列的矩阵，它的形状为 (n, m)
ndarray.dtype	数组中元素的类型
ndarray.size	数组的整体大小，相当于所有维度相乘得到的结果
ndarray.context	数组所在的设备位置，如 <code>cpu</code> 或者 <code>gpu(1)</code>

Symbol 是定义计算图的基本元素，计算图是众多开源框架完成模型训练的基础。我们通过 Symbol 构建出计算图之后，可以对计算图进行编译，然后执行计算图。使用计算图的形式可以提前安排好所有的计算内容，这样也方便系统对其进行优化。

Module 是对部分运算的封装，这些封装而成的模块通常具有一定的通用性，能够在很多场景下使用。下面的代码展示了一个 Module 的创建过程。

```
mod = mx.mod.Module(symbol=net,
                     context=mx.cpu(),
                     data_names=['data'],
                     label_names=['softmax_label'])
```


其中的参数名称及其含义如表 8-4 所示。

表 8-4 创建 Module 的参数名称及其含义

参数名	含义
symbol	网络的定义变量
context	用于运行的设备
data_names	网络中输入数据的名字
label_names	网络中输入标签的名字

通过上面的定义，我们只需要使用很简单的方法就可以完成一些琐碎的操作。

```
# 模型训练
mod.fit(train_iter,
        eval_data=val_iter,
        optimizer='sgd',
        optimizer_params={'learning_rate':0.1},
        eval_metric='acc',
        num_epoch=8)

# 模型预测
y = mod.predict(val_iter)

# 模型评估
score = mod.score(val_iter, ['mse', 'acc'])
```

8.6.4 小试牛刀

以下代码来自 MXNet 官方示例^[6]，实现一个针对 CIFAR-10 的图像识别模型。

```
import os
import argparse
import logging
logging.basicConfig(level=logging.DEBUG)
from common import find_MXNet, data, fit
from common.util import download_file
import MXNet as mx

# 下载数据集
```

```

def download_cifar10():
    data_dir="data"
    fnames = (os.path.join(data_dir, "cifar10_train.rec"),
              os.path.join(data_dir, "cifar10_val.rec"))
    download_file('http://data.MXNet.io/data/cifar10/cifar10_val.rec', fnames
    [1])
    download_file('http://data.MXNet.io/data/cifar10/cifar10_train.rec', fnames
    [0])
    return fnames

if __name__ == '__main__':
    (train_fname, val_fname) = download_cifar10()

    parser = argparse.ArgumentParser(description="train cifar10",

    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    fit.add_fit_args(parser)
    data.add_data_args(parser)
    data.add_data_aug_args(parser)
    data.set_data_aug_level(parser, 2)
    # 这里设置模型的一些参数
    parser.set_defaults(
        network = 'resnet',
        num_layers = 110,
        data_train = train_fname,
        data_val = val_fname,
        num_classes = 10,
        num_examples = 50000,
        image_shape = '3,28,28',
        pad_size = 4,
        batch_size = 128,
        num_epochs = 300,
        lr = .05,
        lr_step_epochs = '200,250',
    )
    args = parser.parse_args()

```

```
# 通过配置参数的方式构建网络
from importlib import import_module
net = import_module('symbols.'+args.network)
sym = net.get_symbol(**vars(args))

# 训练网络
fit.fit(args, sym, data.get_rec_iter)
```

从代码中可以看出,对于常见的模型,MXNet已经给出了很好的框架实现,只需要填入一些参数就可以得到模型。这样极大地方便了用户的使用。

8.7 Keras

除以上提到的深度学习框架之外,还有很多非常优秀的其他选择。而 Keras 则与以上平台存在较大差异,它是一个更高层的 API,可以使用其他平台作为其后端。

8.7.1 概述

Keras^[7]是一个由纯 Python 编写的高层神经网络 API,默认使用 TensorFlow 作为后端计算,也可以切换到 Theano。可以说,Keras 是为支持快速实验而生的,能够快速将想法落地,可以说是上手深度学习的利器。

Keras 具有如下特点。

- 高度模块化。Keras 具有网络层、损失函数、优化器、初始化策略、激活函数和正则化方法等独立模块,搭建一个深度学习模型就像小孩搭积木一样简单。
- API 极简。Keras 具有简单、一致的 API 接口,普通的应用不需要编写多少代码即可完成。
- 易扩展性。如果需要在 Keras 基础上添加新模块,只需要照猫画虎编写新的类或函数即可。
- 纯 Python 编写。Keras 和 Theano 一样,没有单独的模型配置文件类型,模型使用 Python 代码编写,这种方法对于不熟悉程序开发的用户稍微麻烦一些,但对于深度开发者而言,可以更方便进行调试。

8.7.2 安装

可以使用 PyPI 的方式来安装 Keras:

```
sudo pip install keras
```

也可以从源代码一步步自行安装。

(1) 首先安装如下依赖库。

- NumPy、SciPy
- PyYAML
- Theano 或 TensorFlow (可选, 看自己愿意选择哪个基础框架)
- HDF5、H5PY (可选, 仅在模型的 save/load 函数中使用)
- cuDNN (可选, 但如果使用 CNN 之类的操作则建议安装)

(2) 通过 cd 命令进入 Keras 的文件夹中, 并运行下面的安装命令。

```
sudo python setup.py install
```

8.7.3 模块介绍

Keras 的结构比较简单、清晰, 整体主要由以下一些模块构成。

- Layers: 构建模型的一个个模型层。
- Losses: 模型损失函数。
- Metrics: 计算模型精确率。
- Optimizers: 训练优化算法。

还有其他一些模块, 这里就不再介绍了。关于它们的使用方式, 可以从下面的样例中看出。

8.7.4 小试牛刀

下面就来看看官方^[7]给出的关于 MNIST 数据集识别问题的代码。

```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

batch_size = 128
num_classes = 10
epochs = 12

img_rows, img_cols = 28, 28

(x_train, y_train), (x_test, y_test) = mnist.load_data()

# 考虑到Theano和TensorFlow两个框架的数据结构不同, 这里需要根据不同的后端调整数
据维度顺序
if K.image_data_format() == 'channels_first':
    # channels_first的代表: Theano
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    # channels_last的代表: TensorFlow
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
# 将标签数据处理成one-hot型数据
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# 创建一个顺序模型，将所有涉及的计算都顺序放入这个模型中
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

# 构建模型，同时附带模型计算的损失函数、优化算法、评价函数
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

# 进行训练
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

# 模型评估
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

参考文献

- [1] Theano 官网: <http://deeplearning.net/software/theano/>
- [2] Torch 官网: <http://torch.ch/>
- [3] PyTorch 官网: <http://pytorch.org/>
- [4] Caffe 官网: <http://Caffe.berkeleyvision.org>
- [5] TensorFlow 官网: <https://www.TensorFlow.org/>
- [6] MXNeT 官网: <http://mxnet.io/>
- [7] Keras 官网: <https://keras.io/>
- [8] NumPy 官网: <http://www.numpy.org/>

第 2 部分

计算机视觉篇

9

计算机视觉背景

计算机视觉是指模拟人的视觉机理来获取和处理信息的技术，是人工智能尤其是机器学习技术在视觉领域的具体应用，同时也是数学、物理、生物、工程、心理学、计算机等多个学科交叉形成的研究领域。计算机视觉是人工智能领域近年来发展最为迅猛的一个分支，其中，深度学习在该领域取得了变革性的重大进展。

本章作为第2部分计算机视觉篇的引子，将简单介绍传统计算机视觉和基于深度学习的计算机视觉技术。由于作者资历有限，一些久远的视觉历史主要参考了李飞飞老师在CS231n课程上的介绍^[1]。

9.1 传统计算机视觉

随着互联网和智能手机等相关技术和产品的发展，网络上的视频和图像资料多到无法想象，这些资料中蕴含着大量的信息，数据量之巨大，单靠人眼是不可能完全浏览并对数据进行分类整理的。大公司都在想办法对这样的数据进行标记、分类和索引等处理，使得这些信息可以用来帮助投放广告、检索等业务，这时就必须借鉴计算机视觉技术^[1]。

计算机视觉就是能够对图片进行标注、分类等处理的技术，比如识别视频某一帧中的内容。思科的白皮书^[2]做了一个统计，2016年手机视频占到互联网上的流量的60%，这个数字在2021年将达到78%，再加上图片，互联网中85%以上的信息都是视频或图片。这么大的数据量也必然推动计算机视觉的快速发展，其中，深度学习是毋庸置疑的核心技术。

五亿四千万年前,地球处于寒武纪时期,生物开始“进化爆炸”(寒武纪生命大爆发),地球上当时到处是水,所有生物都是瞎子,没有眼睛,它们进食的方式就是张着嘴漂着,等着食物撞进来,然后吞下去。但是当时发生了一些奇怪的事情,从化石研究来看,物种数量突然之间爆发,生物变得多样化,出现了肉食动物,而其他生物也进化来自我逃生的本领。到底是什么力量触发了这一切?

这方面有很多猜测,包括小行星撞地球之类的,其中最有说服力的理论来自于澳大利亚的地质学家 Andrew Parker^[3],他通过研究化石得出结论:这一切都是源于眼睛的出现。

第一个先驱进化出了非常简陋的眼睛,和针孔相机差不多,只能捕捉到光线,观察到一点点环境的变化。但这个本领很强大,生活一下子滋润了,因为这对寻找食物来说太有用了,不再需要漂在水里等吃的。而猎物们需要做的也是尽快进化出眼睛,逃离克星。因此,这个时期第一个进化出眼睛的动物很幸运,拥有一段非常美好的猎食时间。但好景不常在,生物们展开了“装备竞赛”,在这个时期也出现了肉食动物。这就是五亿四千万年前,生物视觉出现的年代,也是进化大爆发的主要驱动力。

视觉领域的一个突破是在工程技术方面——原始照相机(魔法暗箱),在文艺复兴时期,由达芬奇发明。它的大致构造是一面留有小孔的密闭箱,光线进来后会在暗箱内壁形成颠倒且两边相反的影像。但这还不是关于照相机的最早记载,公元前5世纪中国的墨子就发现了小孔成像现象。亚里士多德也谈及了这种现象。此后,才有了电影之类的发明,然后有了摄像机。

视觉领域的另一个重要突破是视觉原理上的研究成果:生物的大脑是如何处理视觉信息的?目前我们知道大概用了五亿四千万年进化出了人类神奇的视觉系统,那人的视觉系统是如何工作的?哈佛大学进行了一项重要研究,当时由两位年轻的博士后 Hubel 和 Wiesel 主持,具体实验为^[4]:

他们先把猫放在屋子里,然后给猫看一些东西,通过插入脑中的电极观察神经元是否被激活。比如他们给猫看一张鱼的图片,观察神经元是否能被激活。他们给猫看了鱼、老鼠、鲜花之类的图片,结果全都没用,猫的基础视觉区没什么反应,捕捉不到任何脉冲。当时他们确实很沮丧。但是,也有好消息,当时用的是幻灯片投影,他们发现在切换幻灯片时神经元被激活了。也就是说,鱼、老鼠、鲜花都没有激活神经元,但是幻灯片切换时反而激活了神经元——切换幻灯片时生成了一个“边缘”,而移动边缘就会激活视觉单元。

他们最终发现视觉皮层的神经元是一列一列组织起来的,每一列神经元只喜欢某一种特定的形状或者某些简单的线条组合,而不是鱼、老鼠、鲜花。他们的研究发现获得了1981年的诺贝尔生理学或医学奖。Hubel 和 Wiesel 的发现表明^[4]:视觉的前期,并不是对鱼或老鼠进行整体识别,而是对简单的形状结构进行处理,这种简单的形状结构就是边缘。

实际上,计算机视觉的很多特征都是围绕边缘进行的,就算是使用深度学习模型,低层也是在抽取一些比较简单的边缘特征,高层则是比较抽象偏整体的特征。

所以,视觉皮层工作机理的发现是第二个非常重要的视觉工作。

那么,计算机视觉起源于什么时候呢?这个可以追溯到现代计算机视觉的先驱——Larry Roberts 在 1963 年写的论文——*Block world* (块状世界)。Hubel 和 Wiesel 的发现表明:边缘决定结构,形状变了,但边缘未变。Larry Roberts 作为早期的计算机博士,他的工作是从图像中解析出边缘和形状。现在回头看这个工作比较简单,大概相当于一个本科生的大作业或毕业设计,但这确实是计算机视觉开天辟地的博士论文。传奇的是, Larry Roberts 后来放弃了计算机视觉的研究,进入 DARPA (Defense Advanced Research Projects Agency, 美国国防部先进研究项目局)。ARPANET 就是 Internet 的前身。他放弃视觉研究后依然做得很好。

但是大家一致认为,计算机视觉真正诞生于 1966 年的夏天,当时 Marvin Minsky 率领的 MIT 人工智能 (Artificial Intelligence, AI) 实验室准备花一个夏天搞定计算机视觉问题,暑期项目就是要设计一套视觉系统:让学生在电脑前面连一个摄像头,然后写一个程序,让计算机告诉我们摄像头看到了什么。虽然那个夏天并未搞定,但是从那以后,计算机视觉成了计算机领域增长最快的方向,比如计算机视觉的顶级会议 CVPR 或者 ICCV 都有 2000 多人参加,自动驾驶、图像识别等方向都是炙手可热的。

计算机视觉领域的另一个大牛 David Marr (天妒英才,1980 年就去世了,年仅 35 岁),当时也在 MIT,他写了一本非常有影响力的书——*Vision*。David Marr 本身是神经科学家和生理学家,他从神经科学领域领悟到很多。Hubel 和 Wiesel 给了我们一个简单的认识:视觉处理流程是从一些简单形状开始的,而不是整体的鱼或老鼠。David Marr 给了我们第二个非常重要的认识:视觉是分层的。

如图 9-1 所示,输入图像是原始像素,第一层得到一些简单的边缘,然后得到 2.5D 结构,这里加入了层次信息,最后得到真正的 3D 结构^[5]。

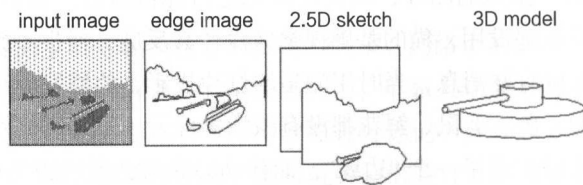


图 9-1 视觉表示的不同层次^[5]

这里有一个问题,自然界是 3D 的,而图像成像是 2D 的,那么人的视觉系统是如何解决这个问题?两只眼睛。事实上,简单的 3D 摄像机就是用两个镜头,然后把图片叠加实

现的。同样，对于语音问题，自然界的语音是重叠的，解决办法是用两只耳朵，在技术上就可以采用多个麦克风。

所以，神经科学带来了两个非常重要的认识：视觉是从简单形状开始的；视觉是分层的。这两个认识也是深度学习架构的最初思想来源，指引大家了解深度学习的学习过程，也指出了推理的过程，但是这并不能帮助大家推导出新的数学公式或模型。

Hubel 和 Wiesel 告诉我们视觉从简单形状开始，David Marr 告诉我们视觉要分层，但是没有人说要建立一个卷积神经网络。

在 20 世纪 80 年代，计算机视觉最好的成果当属 David Lowe 关于简单边缘和形状的相关工作^[6]。

90 年代，终于开始处理彩色图片了。另一个重大发现是不再去识别图片中的物体，而是将图片分割成有意义的几个部分^[7]。对于人来说，当进到一个房间后，不是看到像素，而是看到椅子、桌子和人等，这个称作感知分组。感知分组是视觉领域最重要的问题，不论是生物视觉还是计算机视觉，如果解决不好，就无法深入理解视觉。当然，计算机视觉目前还没有解决。

在视觉领域另一项非常经典的工作是 Viola 和 Jones 所进行的人脸识别^[8]。这个研究成果没有使用深度学习，但是有很强的深度学习特质，算法试图寻找一些黑白的过滤器特征值，更好地实现人脸定位。这也是第一个在电脑上实时运行的计算机视觉方面的研究成果，以前的视觉算法都超慢，这项工作本身就是“实时人脸检测”，可以在很慢的奔腾 II CPU 上实时运行。

当然，Viola 和 Jones 所进行的人脸识别并非 90 年代唯一的成果，但是这个成果反映了视觉领域焦点的迁移：从 3D 重建变成了识别是什么。计算机视觉重新回归人工智能，到目前为止，计算机视觉依然是聚焦识别，只是多了一些与识别相关的定位、检测等外延。

另一个重要成果是关于特征的，在进行人脸识别的那个时期，都希望通过描述整个物体来识别。但实际在现实中有很多遮挡的问题，对人来说，很多遮挡并不是问题，比如看到部分虎纹就知道是老虎，这说明对视觉来说特征可能是最重要的。这次又是 David Lowe 提出了尺度不变特征变换（Scale Invariant Feature Transform, SIFT）算法^[9]，该算法认为只需要看到部分即可进行识别。而且就算整体上有些形变，具体特征也会保持不变。在特征方面还有一个重要成果是空间金字塔匹配模型（Spatial Pyramid Matching, SPM），这在目前的深度学习框架中依然可以找到很多踪影。

利用这些特征，结合机器学习算法，计算机视觉的研究进展非常顺利，当时的机器学习算法主要集中在图像建模方面或者 SVM 上。

在深度学习之前最后一个重要成果是可形变部件模型（Deformable Part Model, DPM），比如一个人，可以由不同的部分组成，这些部件本身也可移动或形变。DPM 连续获得 VOC（Visual Object Class）2007—2009 年的检测冠军，2010 年其作者 Felzenszwalb Pedro 被 VOC 授予“终身成就奖”。DPM 把物体看成了多个组成部件（比如人脸上的鼻子、嘴巴等），用部件间的关系来描述物体，这个特性非常符合自然界中很多物体的非刚体特征。DPM 可以被看作 HOG+SVM 的扩展，它很好地继承了两者的优点，在人脸检测、行人检测等任务上取得了不错的效果。但是 DPM 相对复杂，检测速度也较慢，因此也出现了很多改进的方法。

9.2 基于深度学习的计算机视觉

正当大家热火朝天地改进 DPM 性能的时候，基于深度学习的目标检测横空出世（以 2012 年 ImageNet 竞赛为时间起点），迅速盖过了 DPM 的风头，很多之前研究传统目标检测算法的研究者也开始转向深度学习。

计算机视觉转向深度学习，benchmark 的发展也不容小视。早在 2009 年，计算机视觉就已经足够成熟，大家已经在开始解决一些重要而且复杂的问题了，例如行人识别或汽车识别等，这时候就需要一个 benchmark，否则大家发表论文就会各用各的数据。一个重要的 benchmark 就是 PASCAL VOC，总共 20 个类别、几万张图片。此外，就是 ImageNet 数据集^[10]，这个数据集采用 AWS 众包标注完成，共计 14,197,122 张图片。自 2005 年开始，每年都有相关竞赛 ILSVRC（ImageNet Large Scale Visual Recognition Challenge），数据集来自于 ImageNet，但是只抽取了 1000 个种类、100 多万张图片。有了庞大的数据集，再加上高效的算法和高度并行的 GPU 芯片，深度学习才得以在 ILSVRC2012 中一举夺魁。此后，深度学习则一路高歌猛进，如图 9-2 所示为 ILSVRC 历年 Top-5 错误率。

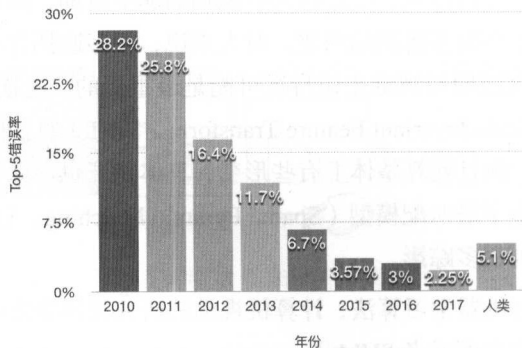


图 9-2 ILSVRC 历年 Top-5 错误率^[10]

9.3 参考文献

- [1] CS231n: Convolutional Neural Networks for Visual Recognition. <http://cs231n.github.io/>.
- [2] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016-2021. White Paper. <http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html>.
- [3] Parker, Andrew (2003). In the Blink of an Eye: How Vision Sparked the Big Bang of Evolution. Cambridge, MA: Perseus Pub.
- [4] Hubel D H, Wiesel T N. Receptive fields of single neurones in the cat's striate cortex[J]. The Journal of physiology, 1959, 148(3): 574-591.
- [5] Marr D. A theory for cerebral neocortex. Proc. R. Soc. Lond., B, Biol. Sci. 176 (43): 161-234, 1970.
- [6] DG Lowe. Perceptual Organization and Visual Recognition. Kluwer Academic Publishers, Boston. 1985.
- [7] Shi J, Malik J. Motion segmentation and tracking using normalized cuts[C]. Computer Vision, 1998. Sixth International Conference on. IEEE, 1998: 1154-1160.
- [8] Viola P, Jones M. Rapid object detection using a boosted cascade of simple features[C]. Computer Vision and Pattern Recognition, 2001. CVPR 2001.
- [9] Lowe D G. Object recognition from local scale-invariant features[C]. Computer vision, 1999. The proceedings of the seventh IEEE international conference on. Ieee, 1999, 2: 1150-1157.
- [10] ImageNet Large Scale Visual Recognition Competition. <http://www.image-net.org/challenges/LSVRC/>.

10

图像分类模型

上一章提到 ILSVRC Top-5 错误率被持续刷新，伴随着这些纪录的刷新，也诞生了很多经典的视觉分类模型，包括 2012 年让深度学习一举成名的 AlexNet^[1]，以及不断刷新纪录的 VGGNet^[2]、GoogLeNet^[3]、ResNet^[4] 等。这些经典的视觉分类模型虽然形状、深度各异，但核心都是卷积神经网络（CNN）模型，因此本章将从最早的 CNN 模型 LeNet-5^[5] 讲起。

10.1 LeNet-5

1998 年 LeCun 发表了用于手写字符识别的经典卷积网络模型 LeNet-5^[5]。当年美国很多银行就采用了这种模型来识别支票上面的手写数字，能够达到如此高的商用程度，其识别精准和在深度学习发展史上的重要地位可见一斑。

LeNet-5 在论文中被用于识别 MNIST 数据集提供的 0~9 共 10 个手写数字。MNIST 是当时 Google 实验室的 Corinna Cortes 和纽约大学柯朗研究所的 Yann LeCun 共同建立的手写数字数据库，包含训练集 60000 张及测试集 10000 张手写数字图片。该数据集提供的图片尺寸统一为 28×28 ，图中包含字符的最大尺寸为 20×20 。

网络输入是一张 $32 \text{ 像素} \times 32 \text{ 像素}$ 的图片，这一尺寸明显大于 MNIST 提供的字符尺寸。这样设计的原因是希望如笔画末端或尖角等潜在的有效特征能够出现在最高层特征检测子感受野的中心位置，换句话说，网络高层的各个特征都集成了笔画和尖角等边缘信息。在 LeNet-5 中，最后一个卷积层（C3）的所有感受野中心会在输入的 $32 \text{ 像素} \times 32 \text{ 像素}$ 图片中形成一个 20×20 的区域。此外，还需对输入像素的取值进行归一化：白色背景取值 -0.1 ，

黑色前景取值 1.175, 这样使得在 MNIST 数据集上像素点取值均值近似为 0, 方差近似为 1。数据归一化能够有效地加速训练过程。

LeNet-5 网络除输入层外共包含 7 层, 每层都含有可学习的参数。如图 10-1 所示, 卷积层标记为 Cx , 下采样层标记为 Sx , 全连接层标记为 Fx , 其中 x 表示层下标。

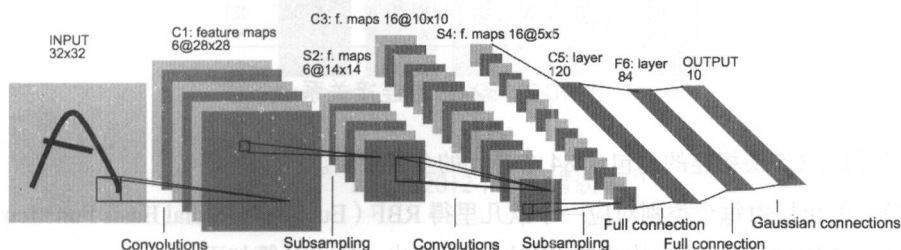


图 10-1 LeNet-5 网络结构^[5]

C1 层是网络的第一个卷积层, 本层需要学习的参数为 6 个尺寸为 5×5 的特征卷积核, 通过这些卷积核输入数据变换为 6 个尺寸为 28×28 ($28 = 32 - 5 + 1$) 的特征图 (Feature Map)。

S2 层是下采样层, 将 C1 层的输出作为输入送至 S2 层的 6 个特征图, 降采样得到的尺寸为 14×14 。S2 层的特征图的每一个单元由 C1 层对应特征图上一个 2×2 的邻域计算而得, 这些邻域互不重叠, 故而 S2 层的特征图的尺寸恰好为 C1 层的一半。在最初的论文^[5]中, 会将邻域里的 4 个值求和乘系数并加上一个偏置, 所得结果再通过 Sigmoid 激活函数来产生 S2 层的特征图。这里系数和偏置便是本层需要学习的参数。在后来的实现中, 常用最大池化 (Max Pooling) 取代上述过程。

C3 层同样是卷积层, 本层卷积核尺寸仍为 5×5 , 通过这些卷积核将输出 16 个尺寸为 10×10 ($10 = 14 - 5 + 1$) 的特征图。由 S2 层的 6 个特征图产生 C3 层的 16 个特征图, 这之间的连接关系有一个比较特别的设计 (如图 10-2 所示): 每个 C3 层的特征图只与部分 S2 层的特征图进行连接。这样做有两个原因:

(1) 部分不完全的连接关系能将连接数控制在一个比较合理的范围内。

(2) 更重要的是, 它强制地打破了网络的对称性, 不同的特征图由于输入不同而能够表达出不同的特征。

S4 层类似于 S2 层, 采用同样的下采样方式得到 16 个尺寸为 5×5 的特征图。

C5 层是网络中最后一个卷积层, 具有 120 个尺寸为 1×1 的特征图。C5 层的特征图的每一个单元将由 S4 层的 16 个特征图分别通过 5×5 的卷积核计算而得。当特征图尺寸为 1×1 时, C5 层等价于一个全连接层。但我们仍以卷积层来命名它, 因为当输入图片尺寸增大时,

网络结构不发生变化，此时特征图的尺寸将超过 1×1 ，故而将 C5 层视为卷积层更有利于模型在使用上的扩展。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X		X			X	X	X
3			X	X	X			X	X	X	X		X		X	X
4				X	X	X			X	X	X	X		X	X	X
5					X	X	X			X	X	X	X		X	X

图 10-2 S2-C3 层连接关系

F6 层是与 C5 层相连的，包含 84 个单元的全连接层。

最后，输出层由每个类别对应一个欧几里得 RBF（Euclidean Radial Basis Function）单元构成。每个 RBF 单元的输入均为 F6 层的 84 个输出，输出计算如下：

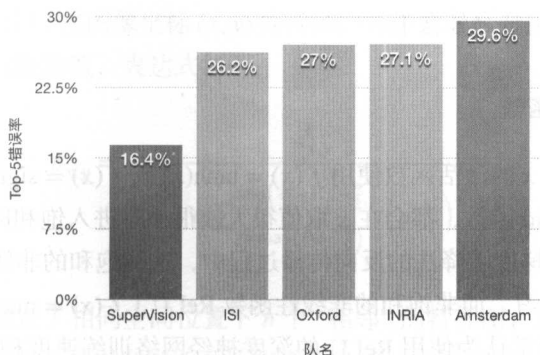
$$y_i = \sum_j (x_i - w_{ij})^2$$

换言之，每个 RBF 单元计算的是输入向量 x 和参数向量 w 之间的欧式距离，输入向量与参数向量之间的距离越远，RBF 单元的输出越大。对于一个特定的 RBF 单元，其输出可以理解成衡量输入向量和对应类别之间匹配程度的一个惩罚项。从概率的观点看，RBF 核的输出也可以理解成在由 F6 层定义的高斯分布下未经归一化的负对数似然。由此，给定一个输入，损失函数将定义为使得 F6 层输出尽量接近期望类别的 RBF 参数向量。采用这种方式，RBF 单元的参数向量需要手动给出，并保持固定（至少在初始的时候），参数向量的每个元素取 -1 或 $+1$ 。

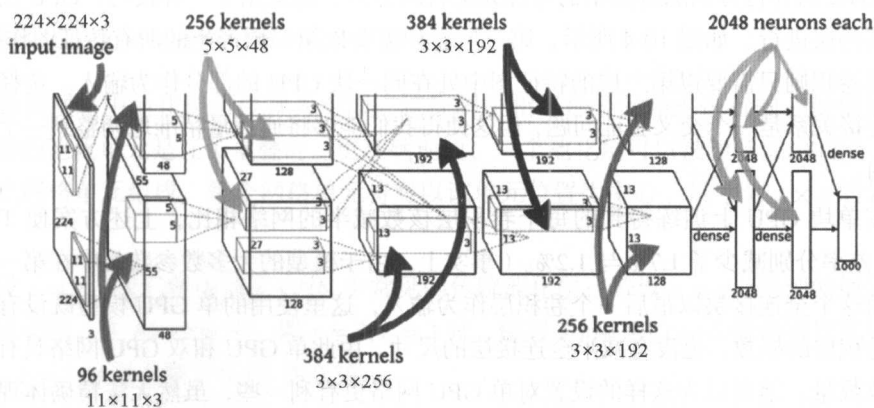
事实上，在后来的实现中，RBF 单元常常被 Softmax 所取代。

10.2 AlexNet

AlexNet^[1] 可以说是一个具有里程碑意义的网络结构。在此之前，深度学习一度陷入沉寂。虽然 2006 年 Geoffrey E. Hinton 等发表的 *Science* 论文^[6] 影响较大，但传统的机器学习势力普遍认为深度学习缺乏理论支持，且在实际中也并未比 SVM 等传统机器学习模型表现更佳。2012 年，Hinton 和他的学生 Alex Krizhevsky 在 ImageNet 的竞赛 ILSVRC2012（1000 个类别、128 万张图片的分类任务）^[7] 中一举刷新了纪录，Top-5 错误率比上一年的冠军下降了十多个百分点，而且远远超过当年的第二名 ISI，如图 10-3 所示（SuperVision 就是 Hinton 团队），从而奠定了深度学习在计算机视觉领域的霸主地位。

图 10-3 ILSVRC2012 Top-5 错误率^[7]

Hinton 团队在这次竞赛中所使用的网络结构被称为 AlexNet，如图 10-4 所示。AlexNet 网络包含 8 个学习层：5 个卷积层和 3 个全连接层，最后的输出层为一个 1000 类的 Softmax 层。AlexNet 模型中间层分为两路，明确显示了两块 GPU 之间的职责划分——一块 GPU 运行图中顶部模型部分，而另一块 GPU 则运行图中底部模型部分。GPU 之间仅在某些层互通通信。该网络的输入是 150,528 维的，且该网络剩下各层的神经元数分别为 253,440—186,624—64,896—64,896—43,264—4096—4096—1000。

图 10-4 AlexNet 网络结构^[1]

AlexNet 的创新和独特之处在于：

1. ReLU 非线性激活函数

通常神经元对输入 x 的激活函数使用 $f(x) = \tanh(x)$ 或 $f(x) = \text{sigmoid}(x) = (1 + e^{-x})^{-1}$ 。无论是 \tanh 还是 sigmoid 函数，都会在 x 取值很大或很小时进入饱和区，这时候神经元的梯度会接近于 0。在使用梯度下降法的反向传播过程中，这些饱和的非线性函数容易造成梯度消失，导致网络很难学习。而非饱和的非线性函数 ReLU ($f(x) = \max(0, x)$) 能够在一定程度上克服这一问题。通常认为使用 ReLU 的神经网络训练速度相比同样使用 \tanh 的网络有数倍提升。

2. 百万数据和多 GPU 训练

随着训练数据的爆发性增长，多 GPU 协同训练更显得日渐重要。AlexNet 当时使用 GTX 580 GPU 进行实验。单块 GTX 580 GPU 只有 3GB 显存，这限制了在其上进行训练的最大网络规模。处理训练模型所需的 128 万张图片数据对于一块 GPU 而言太过吃力，于是 AlexNet 被分拆到了两块 GPU 上。由于 GPU 能够直接对其他 GPU 的显存进行读写，不需要通过主机内存，故而特别适合跨 GPU 并行化。

AlexNet 的并行方案除将模型的神经元进行均分外，还采用了一种技巧：GPU 之间的通信只在某些层进行。如图 10-4 所示，第三层卷积需要以第二层产生的所有特征图作为输入，而第四层卷积则只需要以第三层的特征图中处在同一块 GPU 的部分作为输入。选择层间特征图的连接关系是一个交叉验证问题，但这使得我们能够将通信量精准地调整到一个可接受的范围内。

与在单块 GPU 上训练得到的每个卷积层核数减半的网络相比，上述方案使 Top-1 与 Top-5 误差率分别减少了 1.7% 与 1.2%。（事实上，由于模型的大多数参数集中在第一个全连接层，而这个全连接层以最后一个卷积层作为输入，这里使用的单 GPU 模型既没有减半最后一个卷积层的核数，也没有减半全连接层的尺寸。由此单 GPU 和双 GPU 网络具有大致相同的参数数量。笔者以为这样的设置对单 GPU 网络更有利一些，虽然无法精确体现参数数量对模型精度的影响，但更有效地证明了使用双 GPU 网络的必要性。）训练双 GPU 网络的耗时比单 GPU 网络略少一些。

3. 局部响应归一化 (Local Response Normalization, LRN)

采用 ReLU 一个有利的属性是不需要对输入进行特别的归一化来避免激活函数进入饱和区。对于一个 ReLU 单元，只要训练样本中至少有一部分数据能对其产生正值的输入，这个神经元便能够进行学习。不过，AlexNet 同样提出使用局部归一化有助于模型泛化。

定义 $a_{x,y}^i$ 表示卷积核 i 在图像坐标 (x, y) 处得到的神经元激活值, 继而通过非线性 ReLU, 用 $b_{x,y}^i$ 表示归一化后的激活值, 表达式如下:

$$b_{x,y}^i = \frac{a_{x,y}^i}{\left(k + \alpha \sum_{j=\max(0, i-\frac{n}{2})}^{\min(N-1, i+\frac{n}{2})} (a_{x,y}^j)^2 \right)^\beta}$$

其中, 求和部分涉及了相同空间位置下 n 个“相邻”的特征图, N 表示该层中特征图的总数量。特征图的顺序关系可以在训练开始前任意生成。应用这样的响应归一化, 形成了一种特征图间的侧向抑制, 使不同激活单元的输出值相互竞争。常数 k, n, α, β 是超参数, 通常由验证集决定, AlexNet 当时使用的参数配置是: $k=2, n=5, \alpha=10^{-4}, \beta=0.75$ 。归一化应用于某些层的 ReLU 非线性激活之后。

LRN 这一方案与 Jarrett 等人提出的局部对比度归一化策略有相似之处^[6]。不过, AlexNet 的归一化策略准确说来是一种“亮度归一化”, 它不需要减去平均激活值。采用这一策略, 使得 Top-1 和 Top-5 错误率分别下降了 1.4% 和 1.2%。此外, CIFAR-10 数据集上的实验也同样证明了响应归一化的有效性: 利用一个普通的四层卷积网络, 测试错误率为 13%, 加入归一化后错误率下降到 11%。

4. 重叠池化 (Overlapping Pooling)

在卷积网络中池化层可以视为对同一特征图中相邻神经元输出进行的一种概括。传统上, 相邻的池化单元是互不重叠的。更准确地说, 一个池化层可以被想象成由间隔 s 像素的若干池化网格单元组成, 每个网格单元将对以该单元位置为中心、尺寸为 $z \times z$ 的邻域进行处理。

若设 $s = z$, 我们便得到了 CNN 中常见的传统池化层; 若设 $s < z$, 得到的便是一个有重叠的池化层。在 AlexNet 中, 取 $s = 2, z = 3$ 时, 模型 Top-1 和 Top-5 错误率比采用不重叠的 $s = 2, z = 2$ 分别下降了 0.4% 和 0.3%。

同时在 AlexNet 的训练过程中也观测到, 有重叠的池化层相比传统池化层出现过拟合现象的问题也略有缓解。

5. 整体网络结构

如图 10-4 所示, AlexNet 网络包含 8 个学习层: 5 个卷积层和 3 个全连接层, 最后的输出层为一个 1000 类的 Softmax 层, 产生一个 1000 类标签的分布。网络优化目标是最大化多

分类逻辑回归，也等价于在最大化预测分布下训练样本中正确标签的对数概率平均值。

第二、四、五个卷积层只连接到前一个卷积层中也位于同一块 GPU 上的那些特征图，第三个卷积层则连接着第二个卷积层中的所有特征图，全连接层中的神经元连接前一层所有的神经元。在第一和第二个卷积层后面各接了一个局部响应归一化层。在局部响应归一化层和第五个卷积层之后接的最大池化层采用了前文介绍的重叠池化方式。所有卷积层和全连接层都采用 ReLU 作为非线性激活函数。

网络输入层仍采用 $224 \times 224 \times 3$ 的输入图像。第一个卷积层包含 96 个尺寸为 $11 \times 11 \times 3$ 、步长为 4 个像素的卷积核。第二个卷积层将第一个卷积层的输出作为输入（包含局部响应归一化和池化），该层包含 256 个尺寸为 $5 \times 5 \times 48$ 的卷积核。第三、四、五个卷积层顺序连接，其间不采用池化和归一化。第三个卷积层包含 384 个尺寸为 $3 \times 3 \times 256$ 的卷积核，与第二个卷积层相连（归一化、池化）。第四个卷积层由 384 个尺寸为 $3 \times 3 \times 192$ 的卷积核组成，而第五个卷积层则由 256 个尺寸为 $3 \times 3 \times 192$ 的卷积核组成。每个全连接层都包含 4096 个神经元。

6. 降低过拟合

AlexNet 包含 6000 万个参数。尽管 ILSVRC 的 1000 个类别使得每个训练样本都能够为从图像到标签提供 10bit 约束，但是对学习如此数量庞大的参数模型还是会出现明显的过拟合现象的。

当时 AlexNet 采用的两种对抗过拟合的基本方法如下。

（1）数据增强

很多时候，大规模神经网络的效果直接取决于训练数据的规模，通常充足、丰富的训练数据能够有效地提高网络精度。正因如此，克服过拟合最简单、直接的方法是对标注数据进行一系列变换，并将新产生的数据集作为补充加入训练数据中。其中，通用的图像数据变形包括裁剪、平移、尺度变化、水平翻转，以及增加一些随机的光照、色彩变换。

AlexNet 采用了两种数据增强的变换方式。

第一种变换方式由生成图像裁剪和水平翻转组成。在 256×256 的训练数据原图上随机裁剪尺寸为 224×224 的片段（及它们的水平翻转），并在这些数据上训练网络（这也是图 10-4 中输入图像尺寸为 $224 \times 224 \times 3$ 的原因）。这使得训练集规模扩大了 2048 倍，但显然这样产生的训练样本之间具有高度依赖性。如果不采用这样的策略，网络模型可能过拟合严重，从而迫使我们使用比较小的网络结构。实际中，这种数据增强的方法往往是比较有效的。

在测试阶段，在输入图片的四角和中心共裁剪 5 个 224×224 的图像块及它们的水平翻转都送入网络进行预测，最后对 10 个由 Softmax 层产生的输出进行平均作为最终的预测结果。

第二种变换方式是改变训练数据中 RGB 通道的强度。具体来说，就是对整个 ImageNet 训练数据的 RGB 像素值做主成分分析（PCA）。变换过程是，将一个训练数据的 RGB 叠加上 PCA 得到的主成分所对应的特征向量乘以一个零均值、标准差为 0.1 的高斯分布的随机变量。

这样，RGB 图像的每个像素值为：

$$I_{x,y}' = [I_{x,y}^R, I_{x,y}^G, I_{x,y}^B]^T + [p_1, p_2, p_3] [\alpha_1 \lambda_1, \alpha_2 \lambda_2, \alpha_3 \lambda_3]^T$$

其中， p_i 和 λ_i 表示 RGB 像素值对应的 3×3 协方差矩阵中第 i 个特征向量和特征值， α_i 则是上文中提到的随机系数。采用这种变换能够使网络近似地学习到自然物体识别中一个很重要的属性，即物体识别应该对光照强度和颜色保持不变性。

数据增强策略使 Top-1 错误率降低了超过 1%。

(2) Dropout

结合多个模型的预测结果能有效地降低测试误差，但对于需要花费数天来训练的大型神经网络来说，这样做显得太过昂贵。然而，模型融合有一个非常高效的版本“Dropout”^[8]。“Dropout”以一定的概率（一般取 0.5）将一个神经元的输出设置为 0。以这种方式被关闭的神经元在前向、后向传播过程中都不起作用。这样每次来一个输入，网络便采样出一个不同的结构，但这些结构共享一套参数。由于一个神经元不能只依赖于某些特定的神经元，这种技术降低了神经元间复杂的共适性（Co-Adaptation）。这样，强迫神经元学习更为鲁棒的特征，从而适应与其他神经元的不同随机子集相结合。

在进行测试时，将所有神经元的输出乘以 0.5 来近似一个指数级 Dropout 网络产生预测分布的几何均值。

AlexNet 对前两个全连接层进行 Dropout 操作，这在一定程度上避免了过拟合，但也使得训练收敛所需的迭代轮数增加一倍。

10.3 VGGNet

VGGNet^[2] 和 GoogLeNet^[3] 是 2014 年 ImageNet 竞赛的双雄^[9]，这两类模型结构有一个共同特点是采用了更深的网络结构。当时主要模型的效果对比如图 10-5 所示，其中横坐标表示不同的模型，纵坐标表示测试集上 Top-5 错误率，VGG (paper) 表示论文^[2] 中公布的测试结果。

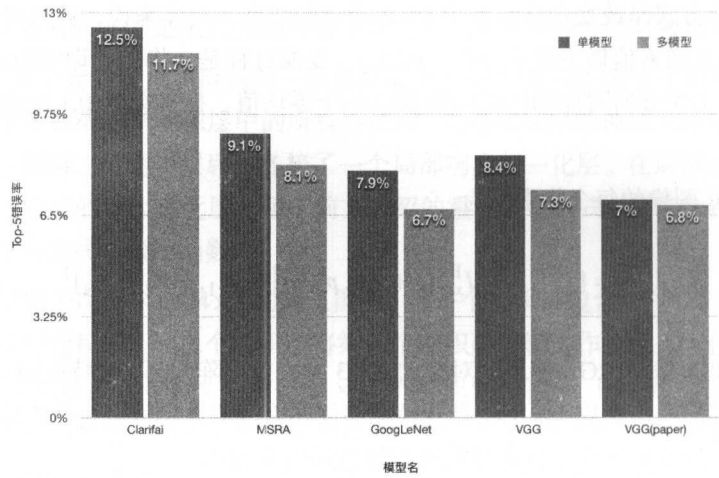


图 10-5 2014 年 ImageNet 竞赛部分结果^[9]

10.3.1 网络结构

如表 10-1 所示，VGGNet^[2] 继承了 AlexNet 的很多结构。

表 10-1 VGGNet 网络结构^[2]

A	A-LRN	B	C	D	E
11 层	11 层	13 层	16 层	16 层	19 层
输入层（224 × 224 RGB 图像）					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
最大池化层					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
最大池化层					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256

续表

A	A-LRN	B	C	D	E
最大池化层					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	conv3-512	conv3-512
					conv3-512
最大池化层					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	conv3-512	conv3-512
					conv3-512
最大池化层					
全连接层-4096					
全连接层-4096					
全连接层-1000					
Softmax 层					

网络输入是一个固定尺寸为 224×224 的 RGB 图像。这里做的唯一预处理是图像需要减去由训练集统计而得的对应像素的 RGB 均值。

预处理之后，图像通过一系列卷积层，VGGNet 采用的卷积核感受野很小： 3×3 （能够包含上、下、左、右邻域信息的最小尺寸）。在其中一组配置中，VGGNet 甚至采用了 1×1 的卷积核，这时卷积退化成为对输入的线性变换（后面跟一个非线性单元）。卷积步长固定使用 1 个像素。

一簇卷积层之后是三个全连接层：前两个各包含 4096 个激活单元，第三个对应 ILSVRC 的 1000 个分类同样采用 1000 个激活单元。最后一层是 Softmax 层。在各组配置中，全连接层的设置都是一致的。

所有的隐层都使用 ReLU 作为激活函数。

在 VGGNet 的实验中，局部响应归一化（LRN）并没有带来性能的提升，却对计算资源和内存都有更多的消耗。因此除一组配置外，其他配置都没有采用 LRN。

10.3.2 配置

表 10-1^[2] 中列出了 VGGNet 实验所采用的几组网络结构配置，其中每一列代表一种配置。后面我们将直接以 A~E 的标号来代表这几组配置。这几组配置的主要区别在于不同的网络深度：从 A 的 11 个参数层（8 层卷积和 3 层全连接）到 E 的 19 个参数层（16 层卷积和 3 层全连接）。卷积层中的特征图数量都相对较少，从第一层的 64 个，每经过一个最大池化层参数就会翻倍，直到最后的 512 个。

表 10-2 列出了各组配置中包含的参数数量，尽管网络深度增加很多，但模型参数并不比有较大卷积感受野和较多特征图的浅层网络多。

表 10-2 VGGNet 网络参数数量^[2]

网络	A, A-LRN	B	C	D	E
参数数量（单位：百万）	133	133	134	138	144

10.3.3 讨论

VGGNet 在整个网络中使用 3×3 的小感受野，以步长 1 进行逐像素卷积。容易看到，网络里有很多两个 3×3 的卷积层级联的小结构，中间没有池化层，实质上具有了尺寸为 5×5 的感受野。同理，三个 3×3 的卷积层级联，便具有了尺寸为 7×7 的有效感受野^[2]。

这种小结构有什么收益呢？我们以三个 3×3 的卷积层级联替代单个 7×7 的卷积层为例进行说明。

首先，这样用整合了的三个非线性激活层替代单一非线性激活层，增加了判别能力。

其次，减少了网络参数。假设三个 3×3 的卷积层的输入输出都是 C 个通道，那么小结构中参数个数为 $3(3^2C^2) = 27C^2$ 。类似地，一个 7×7 的卷积层则需要 $7^2C^2 = 49C^2$ 个参数，多出了 81%。也可以看作对 7×7 的卷积网络施加了某种正则化，迫使其能够分解成三个 3×3 的卷积层（其间还包含非线性激活层）。

使用尺寸为 1×1 的卷积层能够在不影响感受野的情况下，增加网络非线性判别能力。尽管在 VGGNet 中 1×1 卷积实质上是在相同维度空间（输入输出通道数一致）上的线性投影，但激活函数还是引入了额外的非线性能力。

由于使用了小尺寸的卷积核，增加网络深度并不会带来明显的参数膨胀，却能从更深的网络中获得更高的精度。

10.3.4 几组实验

1. 单尺度测试集

表 10-1^[2] 中各个网络结构在单尺度测试集 Q 上的对比实验如表 10-3 所示。其中训练集 S 中每张图片的最短边尺度是在一定区间范围内的（比如 $[256, 512]$ ），表示图片的最短边长度在该区间内随机取值。由于训练数据大小不一，而 VGGNet 要求固定大小的输入，因此必须进一步处理。处理的方法有很多，包括：

- 输入图片全部 **resize** 到固定大小。
- **Scale Jittering**。先固定一种裁剪的尺寸为 $m \times m$ ，比如 224×224 ，然后把图片最短边缩放到一个大于 m 的值，比如 $[256, 512]$ 区间内的任意值，最后再裁剪出 $m \times m$ 的区间。这种数据增强方法使用广泛，当然有时候也会存在问题，比如最后裁剪时可能最重要的部分被裁掉了。VGGNet 中的多尺度数据主要采用这种方法。

从实验可以看到，A~E 随着卷积层的一步加深，准确率的提升也逐渐接近瓶颈。后续有一些论文针对卷积层输入的前处理（如 Batch Normalization）和输出的后处理（如 pReLU）做了其他研究，试图进一步提升效果。但实验表明，Scale Jittering 等数据增强方法对效果的提升是比较明显的。

表 10-3 VGGNet 不同网络配置在单尺度测试集上的对比实验^[2]

VGGNet 配置 (表 10-1)	图像最短边长度		Top-1 错误率 (%)	Top-5 错误率 (%)
	训练集 (S)	测试集 (Q)		
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	$[256, 512]$	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	$[256, 512]$	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	$[256, 512]$	384	25.5	8.0

2. 多尺度测试集

多尺度测试集上的对比实验如表 10-4 所示。考虑到尺度差异过大会导致性能下降，所以测试集 Q 的尺度在训练集 S 上下 32 个像素内浮动。对于训练集是区间尺度的，测试集尺度为区间的最小值、中值及最大值。

表 10-4 VGGNet 不同网络配置在多尺度测试集上的对比实验^[2]

VGGNet 配置 (表 10-1)	图像最短边长度		Top-1 错误率 (%)	Top-5 错误率 (%)
	训练集 (S)	测试集 (Q)		
B	256	224, 256, 288	28.2	9.6
C	256	224, 256, 288	27.7	9.2
	384	352, 384, 416	27.8	9.2
	[256, 512]	256, 384, 512	26.3	8.2
D	256	224, 256, 288	26.6	8.6
	384	352, 384, 416	26.5	8.6
	[256, 512]	256, 384, 512	24.8	7.5
E	256	224, 256, 288	26.9	8.7
	384	352, 384, 416	26.7	8.6
	[256, 512]	256, 384, 512	24.8	7.5

3. 模型融合

VGGNet 原论文^[2]通过平均多个模型后验概率的方式进行模型融合。实验结果表明，融合 D/E 两个模型时效果最好，而融合 7 个模型后效果反而有所下降。

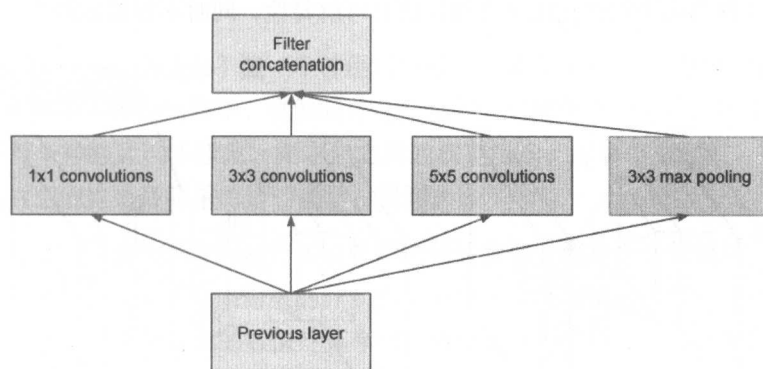
10.4 GoogLeNet

GoogLeNet^[3]是 2014 年 ILSVRC 图像分类和定位两个任务的挑战赛冠军^[9]，用一个 22 层的深度网络将图像分类 Top-5 的错误率降低到 6.67%。为了致敬卷积网络的经典结构 LeNet-5，同时兼顾 Google 的品牌，Google 团队为竞赛模型起了 GoogLeNet 的名字。

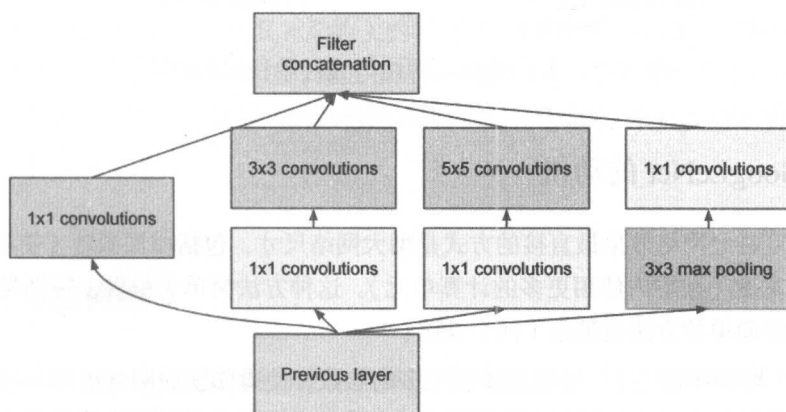
GoogLeNet 通过精巧的网络结构设计，在保持一定计算开销的前提下增加了网络深度和宽度，有效提高了网络内计算资源的利用效率。与两年前的 AlexNet 相比，GoogLeNet 在精度上获得了显著提升，同时将模型参数减少了 12 倍。

为了优化网络质量，GoogLeNet 的设计基于赫布理论和多尺度处理的观点。GoogLeNet 采用了一种高效的机器视觉深度神经网络结构，将其称为“Inception”。在这里，更“深”具有两层含义：一是提出了一种新的网络层组织形式——“Inception Module”；二是直观地增加了网络深度。

在 AlexNet 和 VGGNet 中，全连接层占据 90% 的参数量，而且容易引起过拟合；而 GoogLeNet 用全局平均池化层取代全连接层，这种做法借鉴了 NIN (Network in Network) [10]。此外，其精心设计的 Inception 结构（如图 10-6 所示）也参考了 NIN 的原理。



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

图 10-6 Inception 网络结构^[3]

10.4.1 NIN

NIN^[10]的一个动机是,在传统的CNN中卷积层实质上是一种广义的线性模型,其表达和抽象能力不足,能否使用一种表达能力更强当然也更复杂的子网络代替卷积操作,从而提升传统CNN的表达能力。一种比较简单的子网络就是多层感知机(MLP)网络,MLP由多个全连接层和非线性激活函数组成,如图10-7所示。

相比普通的卷积网络,MLP网络能够更好地拟合局部特征,也就是增强了输入局部的表达能力。在此基础上,NIN不再像卷积一样在分类层之前采用全连接网络,而是采用全局平均池化,这种全局平均池化比全连接层更具可解释性,同时不容易过拟合。

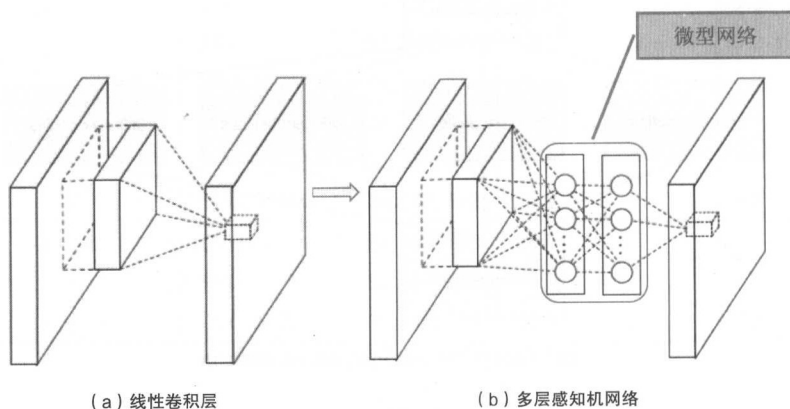


图 10-7 多层感知机网络与线性卷积层的对比^[10]

10.4.2 GoogLeNet 的动机

提高深度神经网络效果最直接的方式是增大网络尺寸,包括增加深度(使用更多的层),也包括增加宽度(在层内使用更多的计算单元)。这种方法简单、稳妥,特别是当有足够标注数据时。但简单的方法也带来了两个缺陷:

(1) 更大的网络尺寸,通常意味着更多的参数,使得膨胀的网络更容易过拟合。在标注数据有限的场景中,这种情况更加明显。类似于ILSVRC这样需要人工甚至专业知识(有的类别很难区分)进行强标注的数据集,增大数据规模是非常昂贵的。这也成为这类思路的主要瓶颈。

(2) 整体增大网络尺寸会显著提高对计算资源的需求。如在深度视觉网络中,两个卷积层级联,如果统一增加卷积核数量,那么计算量的增大将与卷积核数的增加成平方关系。

更坏的情况是，如果新增的网络单元不能发挥足够的作用（如大部分权重最终被优化成0），那么大量的计算资源将被浪费。计算资源是有限的，尽管目标都是提高模型质量，但有效地分配这些资源总是优于盲目扩大网络参数的。

解决这些问题的一个基本方法是引入稀疏性，用稀疏的连接形式取代全连接，甚至在卷积内部也可以这样做。除了模仿生物系统，Arora 等人^[11]通过坚实的理论推导同样开创性地证明了这一优势。他们的主要结论是：如果用一个大型、稀疏的深度神经网络表示某数据集的概率分布，那么最优的网络拓扑可以通过逐层分析与之前神经元的统计相关，并将高相关性的神经元进行聚类得到。尽管严格证明这一观点需要很强的条件，但是我们可以用赫布理论从概念上进行理解。

赫布原则（Hebbin Principle）是一个神经科学理论，解释了在学习过程中脑中的神经元所发生的变化。在人工神经网络中，突触间传递作用的变化被类比为神经元网络中相应权重的变化。如果两个神经元同步激发，则它们之间的权重增加；如果单独激发，则权重减少。赫布原则是最古老的也是最简单的神经元学习规则。

遗憾的是，今天的设备对非一致性稀疏数据结构的数值计算是十分低效的。就算将算数操作降到几百量级，查找和缓存未命中的开销仍为主导——使用稀疏矩阵或许得不偿失。在使用对密集矩阵相乘优化程度高的数值计算库时，情况会变得更糟。尽管存在利用随机稀疏的网络结构来打破对称性的例子，但 AlexNet 主流架构又重新使用全连接网络来优化参数计算，以利用密集计算的高效性。

是否存在一种折中的方法，既具有结构上的稀疏性，又能利用密集矩阵计算呢？在大量稀疏矩阵计算的文献中提到，将稀疏矩阵聚成相对稠密的子矩阵能带来客观的性能提升。不难想象，使用相似的方法自动构建非一致结构的神经网络并不遥远。GoogLeNet 中的 Inception 模块就可以达到此等效果。

10.4.3 网络结构细节

“Inception”结构的主要思想是用便捷可得的密集原件去近似卷积视觉网络的最优局部稀疏结构。接下来需要做的就是找到一种最优的局部结构，重复这种结构把它们拼接在一起组成网络。

如果前一层输出的每个单元都可以认为对应着原始输入图像中的某个区域，那么这些单元共同组成当前的特征图组。在较低层相关单元会集中在局部区域。这样，我们能得到很多集中在一个区域的簇，这些簇形成一个单元并与上一个单元相连。

Arora 等人^[11]提出一种层与层的结构，在结构的最后一层进行相关性统计，将相关性高的单元聚集到一起。这些簇构成下一层的单元，与上一层的单元连接。

假设前面层的每个单元对应于输入图像的某些区域，这些单元被滤波器进行分组。低层（接近输入层）的单元集中在某些局部区域，这意味着最终会得到在单个区域的大量群，它们能在下一层通过 1×1 卷积覆盖^[10]。然而，也可以通过一个簇覆盖更大的空间来减少簇的数量。为了避免 patch-alignment 问题，将滤波器大小限制在 1×1 、 3×3 和 5×5 （主要是为了方便，非必要）。在池化层添加一个备用的池化路径可以提高效率。

由于这些 Inception 模块一层叠一层，它们输出的相关性统计必然有所不同：更高的层用于提取更抽象的特征，空间上的集中程度应该相应地发生退化。这意味着，更高的层 3×3 和 5×5 的卷积核比例就应该更高。

上面模块存在的一个显著问题是（至少在这种朴素模式下），即使 5×5 的卷积核数增加不明显，也会导致最高层的卷积核数激增。如果再加上池化单元，问题会更加严重。

另一种方式是在需要大量计算的地方谨慎进行降维，压缩信息以聚合。

得益于 Embedding 技术的成功，即使低维度的 Embedding 也能包含相对大的图像区域中的丰富信息。然而，Embedding 将信息表达为稠密压缩的模式，处理起来更困难。我们期望的是在大部分地方保持稀疏，只在需要放大的位置产生稠密信号。

于是， 1×1 卷积放置在计算昂贵的 3×3 和 5×5 卷积层前，用于减少计算量。 1×1 卷积不仅用来降维，还用来修正线性特征。

总的来说，将 Inception 模块进行堆叠形成网络。其中一些最大池化层的步长设为 2，减半网格的分辨率。出于技术原因（训练时内存效率），只在高层采用 Inception 模块，底层保持传统卷积层的形式。

对计算昂贵的大图像块卷积先降维。对于这样的框架，一个有用的特性是：增加神经元数量，不会导致下一阶段的计算复杂度显著增加。

此外，这种设计也符合实际操作的直觉：视觉信息需要在不同尺度上进行处理并整合，这样下一阶段就能同时从不同尺度上抽象特征。

对计算资源的优化，可以在不显著增加计算难度的前提下，使得网络每个阶段的宽度和阶段的数量都得以增加。使用精心设计的 Inception 结构网络比相似的非 Inception 结构网络提速 3~10 倍。

如图 10-8 所示为 GoogLeNet 网络详细结构，其中所有 Inception 模块内的卷积都使用 ReLU 作为激活函数，输入为 224×224 的 RGB 图像，数据集处理为零均值。图中“type”表示每一层的具体类型；“patch size”表示卷积层或池化层 kernel 的大小，“stride”为步长；“output size”为输出的大小；“depth”表示各部分内部有几层；“# 3×3 reduce”和“# 5×5 reduce”表示

用于 3×3 和 5×5 卷积层前 1×1 卷积核的数量；“# 1×1 ”“# 3×3 ”“# 5×5 ”分别为对应 kernel 大小的卷积核数量；“pool proj”表示接在 Inception 模块内建的最大池化层后面 1×1 卷积核的数量；“params”为对应参数的数量；“ops”为对应操作的数量（例如乘法）。经验证明，网络结构参数的设置对效果影响不大。参加 ILSVRC 2014 竞赛的 GoogLeNet 使用了 6~7 个模型进行融合。

GoogLeNet 的网络设计充分考虑了计算效率和实用性，因此可以在计算资源、内存受限的独立设备上使用。

type	patch size/ stride	output size	depth	# 1×1	# 3×3 reduce	# 3×3	# 5×5 reduce	# 5×5	pool proj	params	ops
convolution	$7 \times 7/2$	$112 \times 112 \times 64$	1							2.7K	34M
max pool	$3 \times 3/2$	$56 \times 56 \times 64$	0								
convolution	$3 \times 3/1$	$56 \times 56 \times 192$	2		64	192				112K	360M
max pool	$3 \times 3/2$	$28 \times 28 \times 192$	0								
inception (3a)		$28 \times 28 \times 256$	2	64	96	128	16	32	32	159K	128M
inception (3b)		$28 \times 28 \times 480$	2	128	128	192	32	96	64	380K	304M
max pool	$3 \times 3/2$	$14 \times 14 \times 480$	0								
inception (4a)		$14 \times 14 \times 512$	2	192	96	208	16	48	64	364K	73M
inception (4b)		$14 \times 14 \times 512$	2	160	112	224	24	64	64	437K	88M
inception (4c)		$14 \times 14 \times 512$	2	128	128	256	24	64	64	463K	100M
inception (4d)		$14 \times 14 \times 528$	2	112	144	288	32	64	64	580K	119M
inception (4e)		$14 \times 14 \times 832$	2	256	160	320	32	128	128	840K	170M
max pool	$3 \times 3/2$	$7 \times 7 \times 832$	0								
inception (5a)		$7 \times 7 \times 832$	2	256	160	320	32	128	128	1072K	54M
inception (5b)		$7 \times 7 \times 1024$	2	384	192	384	48	128	128	1388K	71M
avg pool	$7 \times 7/1$	$1 \times 1 \times 1024$	0								
dropout (40%)		$1 \times 1 \times 1024$	0								
linear		$1 \times 1 \times 1000$	1							1000K	1M
softmax		$1 \times 1 \times 1000$	0								

图 10-8 GoogLeNet 网络详细结构^[3]

10.4.4 训练方法

本节主要介绍 GoogLeNet 参加 ILSVRC 2014 时的具体训练方法，包括分布式训练方法、训练参数设置、数据增强等。

GoogLeNet 网络采用 DistBelief^[12] 分布式机器学习系统实现了一定的模型和数据并行。论文^[3]中提到网络虽然仅使用 CPU 进行训练，但粗略估计使用若干高端 GPU，模型能够在周内达到收敛，主要的瓶颈是对内存的需求。

模型训练采用异步随机梯度下降法，动量为 0.9^[13]，每 8 个周期学习率下调 4%。

图像采样的方法在竞赛过程的几个月里不断变化，并在已经收敛的模型上进行调整，有时改变诸如 dropout 或 learning rate 这样的超参数，所以很难直接总结出一个最有效的方法。对于一些复杂的问题，一些模型针对较小的相关数据进行训练，一些模型对大数据进行训练。此外，一个在赛后被验证有效的方法是：在 8%~100% 的范围内截取不同尺度、宽高比满足约束 $[\frac{3}{4}, \frac{4}{3}]$ 的图像块进行数据增强。Andrew Howard 提出的光度失真（Photometric Distortion）^[14]能有效克服对训练数据的过拟合。

10.4.5 后续改进版本

随着深度学习相关技术的快速进步，GoogLeNet 后续又产生了几改进版本。

（1）Inception-v2^[15] 在之前的版本中主要加入了 Batch Normalization，相关技术将在本书的第五部分详述；另外也借鉴了 VGGNet 的思想，用两个 3×3 的卷积代替了 5×5 的卷积，不仅降低了训练参数，而且提升了速度。

（2）Inception-v3^[16] 在 v2 的基础上进一步分解大的卷积，比如把 $n \times n$ 的卷积拆分成两个一维的卷积： $1 \times n$ ， $n \times 1$ 。例如 7×7 的卷积可以被拆分为 1×7 和 7×1 两个卷积。此外，采用了一些巧妙的方法进一步优化了部分卷积层的设计。

（3）Inception-v4^[17] 借鉴了 ResNet 可以构建更深层网络的相关思想，设计了一个更深、更优化的模型。

10.5 ResNet

MSRA 的深度残差网络^[4] 在 2015 年 ImageNet 和 COCO 的 5 个领域：ImageNet 识别、ImageNet 检测、ImageNet 定位、COCO 检测以及 COCO 分割取得第一名^[18]。

10.5.1 基本思想

ResNet 解决了超深层 CNN 网络的训练问题，多达 152 层，甚至尝试了 1000 层。

那么更深的网络是否更好？回答这个问题的一大障碍是梯度爆炸/消失。

当更深的网络开始收敛时，“退化”（Degradation）现象便暴露出来：随着网络深度的增加，准确率不出意料地开始饱和，随后便快速下降（如图 10-9 所示）。

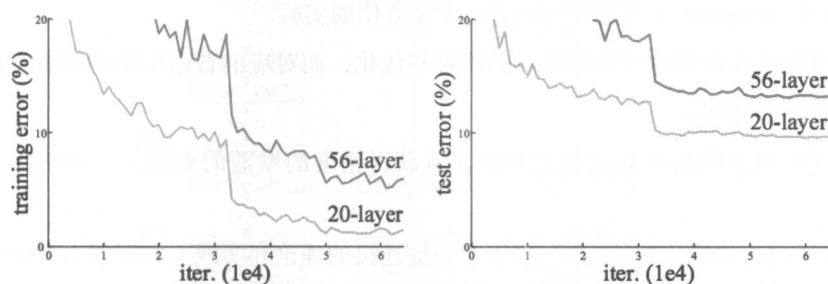
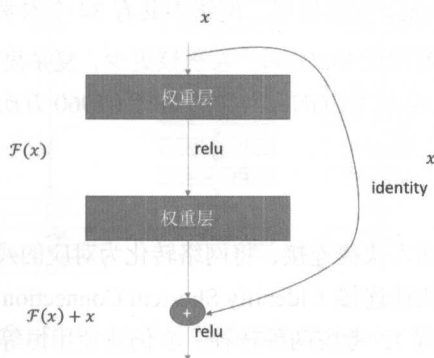


图 10-9 更深的模型同时具有更高的训练误差和测试误差

ResNet 通过引入“深度残差学习”的框架来解决退化问题。ResNet 放弃直接拟合某个函数 $y = H(x)$ 的传统思路,转而拟合残差 $F(x) = H(x) - x$, 原始映射就变成 $H(x) = F(x) + x$ 。ResNet 作者的假设是残差映射比原始映射更容易优化。在极限的情况下,如果优化得到单位映射,那么继续堆叠的非线性层能很容易地拟合出趋向于 0 的残差,而不是再学一个单位映射。

在实际情况中,单位映射不太可能是最优的情况,但是改写为残差能帮助预处理整个问题。当一个最优的函数接近于单位映射而不是零时,找到一个参照单位映射的扰动比学习一个新的函数要更容易。在实验中观察到学到的残差总体上响应很小,也证明了单位映射是一个合理的先验假设。

如图 10-10 所示,公式 $F(x)+x$ 被视为前馈神经网络的“快捷连接”(Shortcut Connection)。快捷连接可以跳过一层或多层,在 ResNet 中快捷连接简单地采用单位映射,并叠加到后面层的输出上。这样既没有增加新的参数,也没有增加计算复杂度。整个网络仍然可以使用 SGD 端到端地进行训练,并很容易地在通用库(如 Caffe)上实现。

图 10-10 残差学习模块^[4]

ResNet 在 ImageNet 上对退化问题进行了全方位的实验。

(1) 即使是非常深的残差网络也仍然容易优化,而对应的普通网络则随着深度的增加表现出更高的训练误差。

(2) 深度残差网络在通过增大网络深度获得精度的增益的表现上,大幅优于先前的网络。

ResNet 采用了快捷连接的方式。关于在快捷连接的理论和实践上的探索由来已久。GoogLeNet 中的 “Inception” 层便是由一个 “快捷” 分支和几个更深的分支构成的。

与 ResNet 同期,高速网络 (Highway Network, HW-Net) [19] 给出了带门函数的快捷连接。

高速网络的初衷类似于 LSTM 的门 (Gate), 都是为了解决由梯度消失带来的信息阻隔问题。高速网络采用了更为 “粗暴” 的方式,在某些门函数上不需要进行转换,而是直接相连把信息传递下去。这样的方式被形象地比喻为额外开辟了一条高速公路。

10.5.2 网络结构

论文[4] 中测试了不同的普通网络和残差网络,它们的具体结构如下。

1. 普通网络

主要受 VGGNet 启发,卷积层主要采用 3×3 的卷积核,并遵循以下两个设计原则。

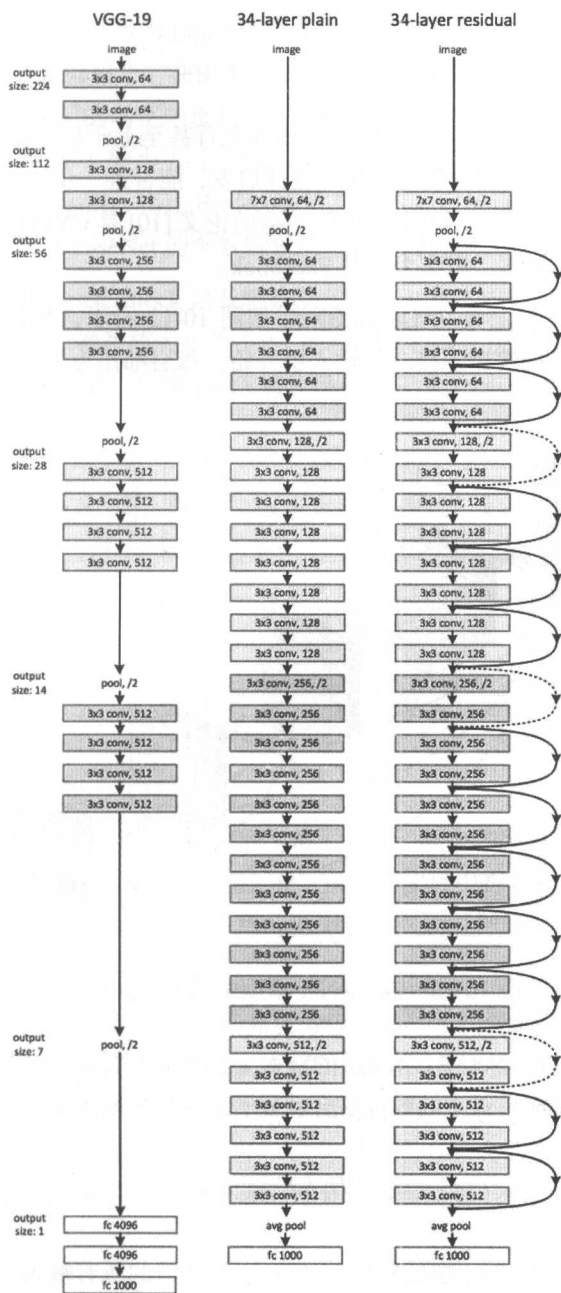
- 如果网络层具有相同数量的特征图输出,那么它们的卷积核数量也一致。
- 如果特征图数量减半,那么卷积核数量就翻倍,从而保持每一层的时间复杂度。

ResNet 通过直接将卷积层的步长设为 2 来实现降采样。网络的最后是一个全局的均值池化层和一个 1000 路的 Softmax 全连接层。网络中共有 34 个参数层。

值得注意的是,这个配置相比 VGGNet,卷积核更少,复杂度更低。34 层的基础网络包含 360 万次乘加操作,仅是 19 层 VGGNet 网络的 18% (1960 万次乘加)。

2. 残差网络

基于上面的普通网络,插入快捷连接,将网络转化为对应的残差版本。当输入输出维度相同时,可以直接采用恒等快捷连接 (Identity Shortcut Connection),如图 10-11 中的右侧实线所示。当维度增加时 (虚线),考虑两种选择:① 仍然使用恒等快捷连接,增加的维度直接补 0,这种方式不会引入新的参数;② 将维度较少的低层输出映射到与高层相同的维度上 (通过 1×1 卷积实现)。无论怎么选择,当快捷连接跨过两层特征图时,步长设为 2。

图 10-11 VGG-19 与 34 层的普通网络及 ResNet 网络的对比^[4]

10.6 DenseNet

ResNet 在一定程度上解决了过深模型（比如几百甚至上千层）梯度发散导致无法训练的问题，其关键之处在于层间的快捷连接。受此启发，能否进一步增加连接，充分利用所有层的特征呢？DenseNet^[20] 就是这样的模型，对应的论文 [10] 是 CVPR 2017 的最佳论文之一，作者分别来自于康奈尔大学、清华大学及 Facebook。

DenseNet 模型中的核心模块 Dense Block 如图 10-12 所示，相比 ResNet 的残差模块，DenseNet 具有更多的跨层快捷连接，从输入层开始，每层都作为后面各层的输入。

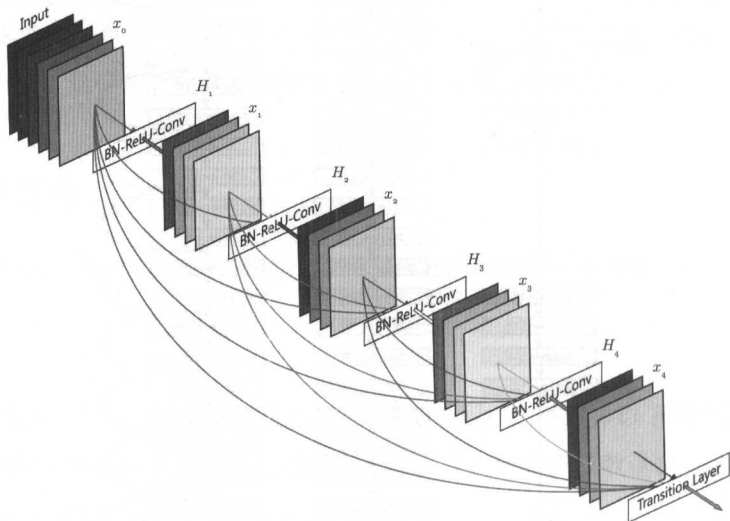


图 10-12 DenseNet 模型中的 Dense Block

在具体实现上，在 ResNet 中，第 l 层的输入 x_{l-1} 经过层的转换函数 H_l 后得到对应的输出 $H_l(x_{l-1})$ ，该输出与输入 x_{l-1} 的线性组合就成了下一层的输入 x_l 。即：

$$x_l = H_l(x_{l-1}) + x_{l-1}$$

而在 Dense Block 中，第 l 层的新增输入 x_{l-1} 与之前的所有输入 $x_0, x_1, \dots, x_{l-3}, x_{l-2}$ 按照通道拼接在一起组成真正的输入，即 $[x_0, x_1, \dots, x_{l-2}, x_{l-1}]$ ，该输入经过一个 Batch Normalization 层、ReLU 和卷积层得到对应的隐层输出 H_l ，该隐层输出就是下一层的新增输入 x_l ，即：

$$x_l = H_l([x_0, x_1, \dots, x_{l-2}, x_{l-1}])$$

x_l 再与之前的所有输入拼接为 $[x_0, x_1, \dots, x_{l-1}, x_l]$ 作为下一层的输入。一般来说，每层新增输入 x_l 的通道数量 k 都很小，在图 10-12 中为 4，原论文中的模型一般取 $k = 12$ 。这个新增通道数量 k 有一个专门的名字叫增长率（Growth Rate）。由于采用这种拼接方式，同时每个隐层特别瘦（即增长率 k 较小），使得 DenseNet 看起来连接很密集，但实际参数数量及对应运算量反而较少。DenseNet 相比 ResNet 在性能上有一定的优势，在 ImageNet 分类数据集上达到同样的准确率，DenseNet 的参数数量及运算量可能只需要 ResNet 的一半左右。

最终的 DenseNet 由 Dense Block 以及转换层（Transition Layer）组成，转换层一般由一个 Batch Normalization 层、卷积核大小为 1×1 的卷积层和池化层组成，其中 1×1 的卷积主要用于瘦身，即降低通道数量。如图 10-13 所示是包含三个 Dense Block 的 DenseNet 模型。

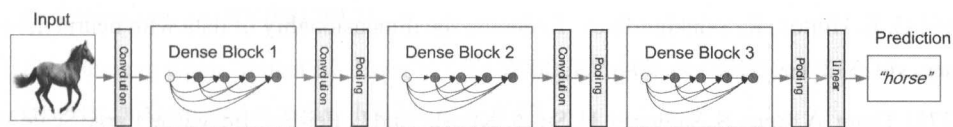


图 10-13 包含三个 Dense Block 的 DenseNet 模型

10.7 DPN

2017 年的 ILSVRC 竞赛（ImageNet）成了最后一届，其中的大赢家无疑是颜水成老师率领的 360 人工智能研究院-NUS 联合实验室团队，在所有项目中都名列前三，其中在目标定位的两项任务中均取得第一的好成绩。所采用的模型称为双通道神经网络（Dual Path Network, DPN）^[21]。

DPN 的原论文^[21] 首先从高阶 RNN 的角度分析了上一节提到的 DenseNet 及其与 ResNet 之间的关系，并发现 ResNet 实质上是 DenseNet 的一种，只是连接为跨层共享。此外，ResNet 通过残差快捷连接隐含地对特征进行了重用，而 DenseNet 则通过稠密连接进行新特征的探索。为了结合两者的优势，作者提出了 DPN，这种模型具有更高的参数有效性、更少的计算量、更少的内存需求，也更容易优化。更多的细节请参考文献 [21]。

参考文献

[1] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106-1114, 2012.

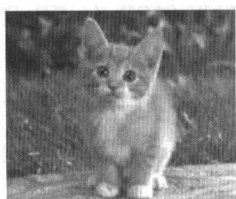
- [2] K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, arXiv technical report, 2014.
- [3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions, 19-Sept-2014.
- [4] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]. Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.
- [5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, November 1998.
- [6] G. E. Hinton, R. Salakhutdinov. Reducing the dimensionality of data with neural networks, Science, vol. 313, no. 5786, pp. 504-507, 2006.
- [7] J. Deng, A. Berg, S. Satheesh, H. Su, A. Khosla, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Competition 2012 (ILSVRC2012). <http://www.image-net.org/challenges/LSVRC/2012/>.
- [8] Srivastava N, Hinton G E, Krizhevsky A, et al. Dropout: a simple way to prevent neural networks from overfitting[J]. Journal of Machine Learning Research, 2014, 15(1): 1929-1958.
- [9] ImageNet Large Scale Visual Recognition Competition 2014. <http://www.image-net.org/challenges/LSVRC/2014/>.
- [10] M. Lin, Q. Chen, and S. Yan. Network in network. CoRR, abs/1312.4400, 2013.
- [11] S. Arora, A. Bhaskara, R. Ge, and T. Ma. Provable bounds for learning some deep representations. CoRR, abs/1310.6343, 2013.
- [12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, NIPS, pages 1232-1240. 2012.
- [13] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton. On the importance of initialization and momentum in deep learning. In ICML, volume 28 of JMLR Proceedings, pages 1139-1147. JMLR.org, 2013.
- [14] A. G. Howard. Some improvements on deep convolutional neural network based image classification. CoRR, abs/1312.5402, 2013.
- [15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of The 32nd International Conference on Machine Learning, pages 448-456, 2015.

- [16] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.. Rethinking the inception architecture for computer vision. In: CVPR (2016).
- [17] Szegedy C, Ioffe S, Vanhoucke V, et al. Inception-v4, inception-resnet and the impact of residual connections on learning[J]. arXiv preprint arXiv:1602.07261, 2016.
- [18] ImageNet Large Scale Visual Recognition Competition 2015. <http://www.image-net.org/challenges/LSVRC/2015/>.
- [19] Srivastava R K, Greff K, Schmidhuber J. Highway networks[J]. arXiv preprint arXiv:1505.00387, 2015.
- [20] G. Huang, Z. Liu, K.Q. Weinberger, and L.J.P. van der Maaten. Densely Connected Convolutional Networks. In: CVPR (2017).
- [21] Yunpeng Chen, Jianan Li, Huaxin Xiao, Xiaojie Jin, Shuicheng Yan, Jiashi Feng. Dual Path Network. arXiv preprint arXiv: 1707.01629, 2017.

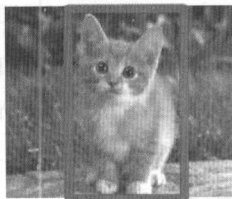
11

目标检测

普通的深度学习监督算法主要用来做分类，如图 11-1 (a) 所示，分类的目标是要识别出图中所示是一只猫。而在 ILSVRC (ImageNet Large Scale Visual Recognition Challenge) 竞赛以及实际的应用中，还包括目标定位和目标检测等任务。其中目标定位不仅仅要识别出是什么物体 (即分类)，而且还要预测物体的位置，位置一般用边框 (Bounding Box) 标记，如图 11-1 (b) 所示。而目标检测实质上是多目标 (目标数量不固定) 的定位，即要在图片中定位多个目标物体，包括分类和定位。比如对图 11-1 (c) 进行目标检测，得到的结果是有几只不同的动物，它们的位置如图中不同大小的框所示。



(a) 目标分类



(b) 目标定位



(c) 目标检测

图 11-1 目标分类、定位、检测示例

简单来说，分类、定位和检测的区别如下。

- 分类：是什么？
- 定位：在哪里？是什么？（单个目标）
- 检测：在哪里？分别是什么？（多个目标）

目标检测对于人类来说并不困难,通过对图片中不同颜色模块的感知很容易定位并分类出其中目标物体。但对于计算机来说,面对的是 RGB 像素矩阵,很难从图像中直接得到狗和猫这样的抽象概念并定位其位置,再加上有时候多个物体和杂乱的背景混杂在一起,目标检测更加困难。但这难不倒科学家们,在传统的视觉领域,目标检测就是一个非常热门的研究方向,对一些特定目标的检测,比如人脸检测和行人检测已经有非常成熟的技术了。普通的目标检测也进行过很多尝试,但是效果总是差强人意。

传统的目标检测一般使用滑动窗口的框架,主要包括三个步骤。

(1) 利用不同尺寸的滑动窗口框住图中的某一部分作为候选区域。

(2) 提取与候选区域相关的视觉特征。比如人脸检测常用的 Harr 特征;行人检测和普通目标检测常用的 HOG 特征等。

(3) 利用分类器进行识别,比如常用的 SVM 模型。

在传统的目标检测中,多尺度形变部件模型 DPM (Deformable Part Model)^[1] 是出类拔萃的。DPM 把物体看成了多个组成的部件(比如人脸的鼻子、嘴巴等),用部件间的关系来描述物体,这个特性非常符合自然界很多物体的非刚体特征。DPM 可以看作 HOG+SVM 的扩展,很好地继承了两者的优点,在人脸检测、行人检测等任务上取得了不错的效果。但是 DPM 相对复杂,检测速度也较慢,因而也出现了很多改进的方法。正当大家热火朝天改进 DPM 性能的时候,基于深度学习的目标检测横空出世,迅速盖过了 DPM 的风头,很多之前研究传统目标检测算法的研究者也开始转向深度学习。

基于深度学习的目标检测发展起来后,其实效果也一直难以突破。比如参考文献[2]中的算法在 VOC 2007 测试集上的 mAP 只有 30% 多一点,参考文献[3]中的 OverFeat 在 ILSVRC 2013 测试集上的 mAP 只能达到 24.3%。2013 年 R-CNN 诞生了, VOC 2007 测试集的 mAP 被提升至 48%, 2014 年通过修改网络结构又飙升到 66%, 同时 ILSVRC 2013 测试集的 mAP 也被提升至 31.4%。

R-CNN (Region-based Convolutional Neural Network, 基于区域的卷积神经网络) 是一种结合区域提名 (Region Proposal) 和卷积神经网络 (CNN) 的目标检测方法。Ross Girshick 在 2013 年的开山之作 *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*^[4] 奠定了这个子领域的基础,这篇论文后续版本发表在 *CVPR 2014*^[5] 上, 期刊版本发表在 *PAMI 2015*^[6] 上。

其实在 R-CNN 之前已经有很多研究者尝试用深度学习的方法来进行目标检测,包括 OverFeat^[3], 但 R-CNN 是第一个真正可以工业级应用的解决方案,这也和深度学习本身的发展类似,神经网络、卷积网络都不是新概念,但在 21 世纪突然真正变得可行,而一旦可行之后再迅猛发展就不足为奇了。

目前对 R-CNN 这个领域的研究非常活跃,先后出现了 R-CNN^{[4][5][6][7]}、SPP-net^{[8][9]}、Fast R-CNN^{[10][11]}、Faster R-CNN^{[12][13]}、R-FCN^{[14][15]}、YOLO^{[16][17]}、SSD^{[18][19]} 等研究。Ross Girshick 作为这个领域的开山鼻祖,像是神一样的存在,R-CNN、Fast R-CNN、Faster R-CNN、YOLO 都和他有关。这些创新的工作其实很多时候是把一些传统视觉领域的方法和深度学习结合起来,比如选择性搜索 (Selective Search) 和图像金字塔 (Pyramid) 等。

与深度学习相关的目标检测方法大致分为两派。

- 基于区域提名的,如 R-CNN、SPP-net、Fast R-CNN、Faster R-CNN、R-FCN。
- 无需区域提名的,如 YOLO、SSD。

目前来说,基于区域提名的方法依然占据上风,但无需区域提名的方法在速度上优势明显,后续的发展拭目以待。

11.1 相关研究

本节作为目标检测的一个回顾,先来看看在目标检测中广泛使用的区域提名——选择性搜索,以及用深度学习进行目标检测的早期工作——OverFeat。

11.1.1 选择性搜索

目标检测的第一步是要做区域提名 (Region Proposal),也就是找出可能的感兴趣区域 (Region Of Interest, ROI)。区域提名类似于光学字符识别 (OCR) 领域的切分,OCR 切分常用过切分方法,简单地说,就是尽量切碎到小的连通域 (比如小的笔画之类的),然后再根据相邻块的一些形态学特征进行合并。但目标检测的对象相比 OCR 领域千差万别,而且图形不规则,大小不一,所以在一定程度上可以说区域提名是比 OCR 切分更难的一个问题。

区域提名可能的方法有:

(1) 滑动窗口。滑动窗口本质上就是穷举法,利用不同的尺度和长宽比把所有可能的大小块都穷举出来,然后送去识别,识别出概率大的就留下来。很明显,这样的方法复杂度太高,产生了很多的冗余候选区域,在现实当中不可行。

(2) 规则块。在穷举法的基础上进行一些剪枝,只选用固定的大小和长宽比。这在一些特定的应用场景中是很有效的,比如拍照搜题 APP 小猿搜题中的汉字检测,因为汉字方方正正,长宽比大多比较一致,因此用规则块做区域提名是一种比较合适的选择。但是对于普通的目標检测来说,规则块依然需要访问很多的位置,复杂度高。

(3) 选择性搜索。从机器学习的角度来说,前面的方法召回率是不错了,但是精度还较低,所以问题的核心在于如何有效地去除冗余候选区域。其实冗余候选区域大多是发生了重叠,选择性搜索利用这一点,自底向上合并相邻的重叠区域,从而减少冗余。

区域提名并不只有以上所讲的三种方法,实际上这个部分是非常灵活的,因此变种也很多,有兴趣的读者不妨参考文献 [20]。

选择性搜索的具体算法细节^[21]如算法 11-1 所示。总体上,选择性搜索是自底向上不断合并候选区域的迭代过程。

算法 11-1 选择性搜索算法

输入: 一张图片

输出: 候选的目标位置集合 L

算法:

1. 利用过切分方法得到候选的区域集合 $R = \{r_1, r_2, \dots, r_n\}$
 2. 初始化相似集合 $S = \phi$
 3. foreach 邻居区域对 (r_i, r_j) do
 4. 计算相似度 $s(r_i, r_j)$
 5. $S = S \cup s(r_i, r_j)$
 6. while $S \neq \phi$ do
 7. 得到最大的相似度 $s(r_i, r_j) = \max(S)$
 8. 合并对应的区域 $r_t = r_i \cup r_j$
 9. 移除 r_i 对应的所有相似度: $S = S \setminus s(r_i, r_*)$
 10. 移除 r_j 对应的所有相似度: $S = S \setminus s(r_*, r_j)$
 11. 计算 r_t 对应的相似度集合 S_t
 12. $S = S \cup S_t$
 13. $R = R \cup r_t$
 14. $L = R$ 中所有区域对应的边框
-

从算法不难看出， R 中的区域都是合并后的，因此减少了不少冗余，相当于提升了准确率，但是别忘了我们还需要继续保证召回率，因此算法 11-1 中的相似度计算策略就显得非常关键了。如果简单采用一种策略，则很容易错误合并不相似的区域，比如只考虑轮廓时，不同颜色的区域很容易被误合并。选择性搜索采用多样性策略来增加候选区域以保证召回率，比如颜色空间考虑 RGB、灰度、HSV 及其变种等，计算相似度时既考虑颜色相似度，又考虑纹理、大小、重叠情况等。

总体上，选择性搜索是一种比较朴素的区域提名方法，被早期的基于深度学习的目标检测方法（包括 OverFeat 和 R-CNN 等）广泛利用，但被当前的新方法弃用了。

11.1.2 OverFeat

OverFeat^{[3][22]} 是用 CNN 统一来做分类、定位和检测的经典之作，其作者是深度学习大神之一——Yann Lecun 在纽约大学的团队。OverFeat 也是 ILSVRC 2013 任务 3（分类 + 定位）的冠军得主^[23]。

OverFeat 的核心思想有三点。

- 区域提名：结合滑动窗口和规则块，即多尺度（Multi-scale）的滑动窗口。
- 分类和定位：统一用 CNN 来做分类和预测边框位置，模型与 AlexNet 类似，其中 1~5 层为特征抽取层，即将图片转换为固定维度的特征向量，6~9 层为分类层（分类任务专用），不同的任务（分类、定位、检测）公用特征抽取层（1~5 层），只替换 6~9 层。
- 聚合：因为用了滑动窗口，同一个目标对象会有多个位置，也就是多个视角；因为用了多尺度，同一个目标对象又会有多个大小不一的块。这些不同位置 and 不同大小块上的分类置信度会进行累加，从而使得判定更为准确，模型的鲁棒性更好。

OverFeat 的关键步骤有四步。

（1）利用滑动窗口进行不同尺度的区域提名，然后使用 CNN 模型对每个区域进行分类，得到类别和置信度，如图 11-2 所示。

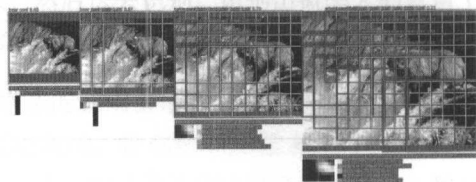


图 11-2 OverFeat 关键步骤一

(2) 利用多尺度滑动窗口来增加检测数量,提升分类效果,如图 11-3 所示。

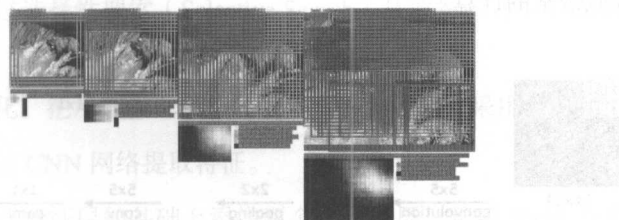


图 11-3 OverFeat 关键步骤二

(3) 用回归模型预测每个对象的位置,从图 11-4 来看,放大比例较大的图片,边框数量也较多。

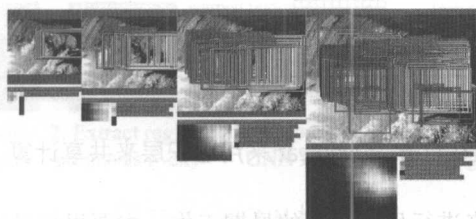


图 11-4 OverFeat 关键步骤三

(4) 边框合并,如图 11-5 所示。

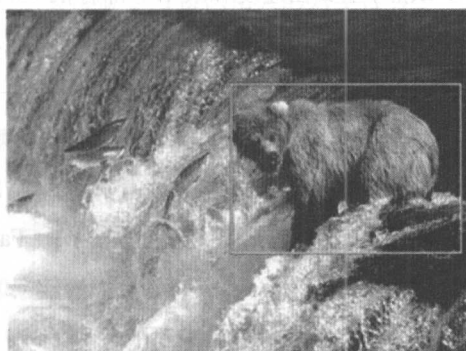


图 11-5 OverFeat 关键步骤四

OverFeat 的另一个重要贡献是它的滑动窗口分类回归非常高效,在此之前,很多滑动窗口技术都是为每个窗口重复进行所有的计算,这对计算资源的消耗是巨大的。而 OverFeat 通过将全连接层改造成卷积层的方式,使得相同区域的计算结果可以共享。如图 11-6 所示,当滑动窗口是 14×14 ,而图片尺寸是 16×16 时,有 4 个滑动窗口需要进行重复计算。OverFeat

采用了卷积计算共享的方式，虽然计算结果是一个滑动窗口的 4 倍，但是计算过程只增加了图中浅灰色区域带来的计算。

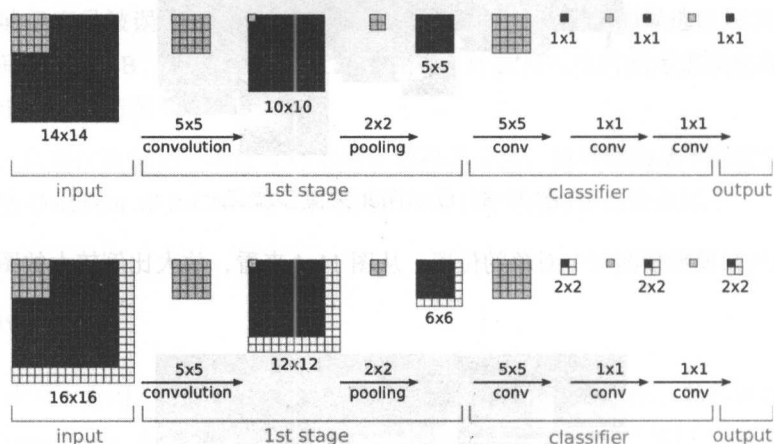


图 11-6 OverFeat 采用卷积层来共享计算

OverFeat 是 CNN 用来进行目标检测的早期工作，主要思想是采用多尺度滑动窗口来做分类、定位和检测，虽然是多个任务，但重用了模型前面几层，这种模型重用的思路也是后来 R-CNN 系列不断沿用和改进的经典做法。

当然，OverFeat 也有不少缺点，至少在速度和效果上都有很大的改进空间，后面的 R-CNN 系列在这两方面做了很多改进。

11.2 基于区域提名的方法

本节主要介绍基于区域提名的方法，包括 R-CNN、SPP-net、Fast R-CNN、Faster R-CNN、R-FCN。

11.2.1 R-CNN

如前面所述，早期的目标检测大都使用滑动窗口的方式进行窗口提名，这种方式本质上是穷举法，R-CNN^{[4][5][6][7]} 采用的是选择性搜索。

以下是 R-CNN 的主要步骤。

(1) 区域提名：通过选择性搜索 (Selective Search) 从原始图片中提取 2000 个左右的区域候选框。

(2) 区域大小归一化：把所有候选框缩放成固定大小 (原文采用 227×227)。

(3) 特征提取：通过 CNN 网络提取特征。

(4) 分类与回归：在特征层的基础上添加两个全连接层，再用 SVM 分类来进行识别，用线性回归来微调边框位置与大小，其中每个类别单独训练一个边框回归器。

其中目标检测系统的结构如图 11-7 所示。注意，图中的第 2 步对应步骤中的 1、2 步，即包括区域提名和区域大小归一化。

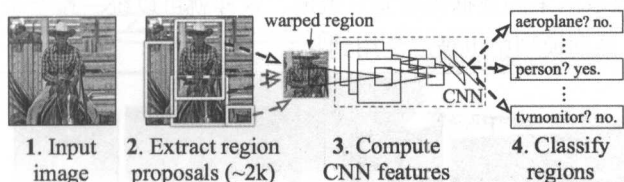


图 11-7 R-CNN 框架^[4]

OverFeat 可以看作 R-CNN 的一个特殊情况，只需要把选择性搜索换成多尺度的滑动窗口，每个类别的边框回归器换成统一的边框回归器，SVM 换为多层网络即可。但是 OverFeat 实际比 R-CNN 快 9 倍，这主要得益于与卷积相关的共享计算。

当然，R-CNN 这次是冲着效果来的，其中 ILSVRC 2013 数据集上的 mAP 由 OverFeat 的 24.3% 提升到 31.4%，第一次有了质的改变。

事实上，R-CNN 有很多缺点。

- 重复计算：R-CNN 虽然不再穷举，但依然有 2000 个左右的候选框，这些候选框都需要进行 CNN 操作，计算量仍然很大，其中有不少其实是重复计算。
- SVM 模型：分类和回归使用 SVM 模型，而且还是线性模型，无法将梯度后向传播给卷积特征提取层，在标注数据不缺的时候显然不是最好的选择。
- 训练测试分为多步：区域提名、特征提取、分类、回归都是断开的训练的过程，中间数据还需要单独保存。
- 训练的空间和时间代价很高：卷积出来的特征需要先存在硬盘上，这些特征需要几百 GB 的存储空间。
- 慢：前面的缺点最终导致 R-CNN 出奇的慢，在 GPU 上处理一张图片需要 13 秒，在 CPU 上则需要 53 秒^[5]。

11.2.2 SPP-net

SPP-net^{[8][9]} 是 MSRA 的何恺明等人提出的,其主要思想是去掉了原始图片上的 crop/warp 等操作,换成了卷积特征上的空间金字塔池化层 (Spatial Pyramid Pooling, SPP),如图 11-8 所示。之所以引入 SPP 层,主要原因是 CNN 的全连接层要求输入图片的大小一致,而实际中的输入图片往往大小不一,如果直接缩放到同一尺寸,很可能有的物体会充满整张图片,而有的物体只占到图片的一角。传统的解决方案是进行不同位置的裁剪,但是这些裁剪技术都可能会导致出现一些问题,比如图 11-8 中的 crop 会导致物体不全, warp 导致物体被拉伸后形变严重, SPP 就是为解决这种问题而产生的。SPP 对整图提取固定维度的特征,再把图片均分成 4 份,每份提取相同维度的特征,再把图片均分为 16 份,依此类推。可以看出,无论图片大小如何,提取出来的数据维度是一致的,这样就可以统一送至全连接层了。SPP 思想在后来的 R-CNN 系列模型中也被广泛用到。

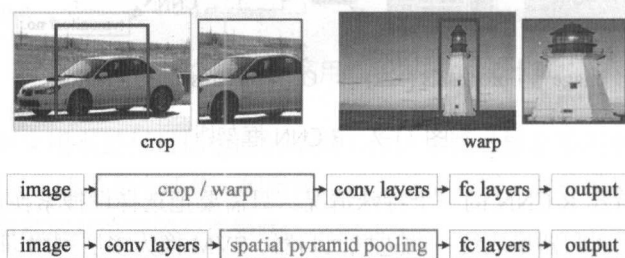


图 11-8 传统 crop/warp 结构和空间金字塔池化网络的对比^[8]

SPP 的优点在于:

- 不论输入图片大小是多少, SPP 都能抽取到固定长度的特征。
- SPP 使用了多级的空间尺度特征, 鲁棒性更好。
- 由于输入维度可变, SPP 能够在不同的维度抽取特征。

SPP-net 的网络结构如图 11-9 所示, 实质上是最后一个卷积层后加了一个 SPP 层, 将维度不一的卷积特征转换为维度一致的全连接输入。

SPP-net 进行目标检测的主要步骤如下。

(1) 区域提名: 用选择性搜索从原图中生成 2000 个左右的候选框。

(2) 区域大小缩放: SPP-net 不再做区域大小归一化, 而是缩放到 $\min(w, h) = s$, 即统一图像长宽尺寸中最短边长度, s 选自 $\{480, 576, 688, 864, 1200\}$ 中的一个, 选择的标准是使得缩放后的目标物体候选框大小与 224×224 最接近。

(3) 特征提取：利用 SPP-net 网络结构提取特征。

(4) 分类与回归：类似于 R-CNN，利用 SVM 基于上面的特征训练分类器模型，用边框回归来微调候选框的位置。

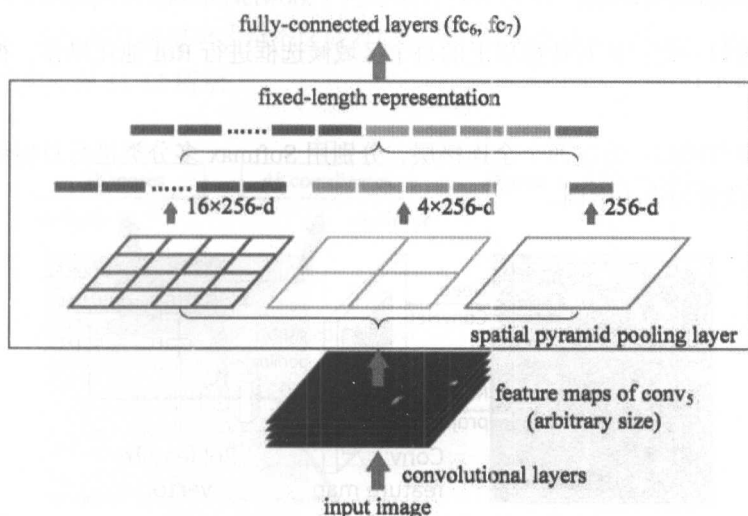


图 11-9 SPP-net 网络结构^[8]

SPP-net 解决了 R-CNN 区域提名时 crop/warp 带来的偏差问题，提出了 SPP 层，网络可以对任意尺度和长宽比的目标物体进行处理，不需要直接对候选框对应图像进行裁剪或拉伸操作，但其他方面依然和 R-CNN 一样，因而仍然存在不少问题，这就有了后面的 Fast R-CNN。

11.2.3 Fast R-CNN

Fast R-CNN^{[10][11]} 是要解决 R-CNN 和 SPP-net 2000 个左右候选框带来的重复计算问题，其主要思想为：

- 使用一个简化的 SPP 层——RoI (Region of Interest) 池化层，操作与 SPP 类似。
- 训练和测试时不再分多步：不再需要额外的硬盘来存储中间层的特征，梯度能够通过 RoI 池化层直接传播；此外，分类和回归用 Multi-task 的方式一起进行。
- SVD：使用 SVD 分解全连接层的参数矩阵，压缩为两个规模小很多的全连接层。

如图 11-10 所示，Fast R-CNN 的主要步骤如下。

- (1) 特征提取：以整张图片为输入，利用 CNN 得到图片的特征层。
- (2) 区域提名：通过选择性搜索等方法从原始图片中提取区域候选框，并把这些候选框一一投影到最后的特征层。
- (3) 区域归一化：针对特征层上的每个区域候选框进行 RoI 池化操作，得到固定大小的特征表示。
- (4) 分类与回归：通过两个全连接层，分别用 Softmax 多分类进行目标识别，用回归模型进行边框位置与大小微调。

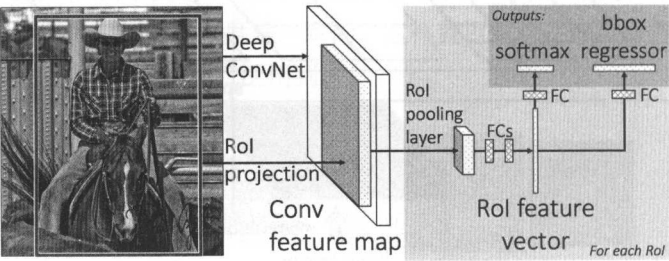


图 11-10 Fast R-CNN 框架^[10]

如图 11-11 所示，Fast R-CNN 比 R-CNN 的训练速度（大模型 L）快 8.8 倍，测试速度快 213 倍；比 SPP-net 训练速度快 2.6 倍，测试速度快 10 倍左右。

	Fast R-CNN			R-CNN			SPPnet
	S	M	L	S	M	L	†L
train time (h)	1.2	2.0	9.5	22	28	84	25
train speedup	18.3×	14.0×	8.8×	1×	1×	1×	3.4×
test rate (s/im)	0.10	0.15	0.32	9.8	12.1	47.0	2.3
▷ with SVD	0.06	0.08	0.22	-	-	-	-
test speedup	98×	80×	146×	1×	1×	1×	20×
▷ with SVD	169×	150×	213×	-	-	-	-
VOC07 mAP	57.1	59.2	66.9	58.5	60.2	66.0	63.1
▷ with SVD	56.5	58.7	66.6	-	-	-	-

图 11-11 Fast R-CNN、R-CNN 和 SPP-net 的运行时间比较^[10]

11.2.4 Faster R-CNN

Fast R-CNN 使用选择性搜索来进行区域提名，速度依然不够快。Faster R-CNN^{[12][13]} 则直接利用 RPN（Region Proposal Network）网络来计算候选框。RPN 以一张任意大小的图片为输入，输出一批矩形区域提名，每个区域对应一个目标分数和位置信息。Faster R-CNN 中的 RPN 网络结构如图 11-12 所示。

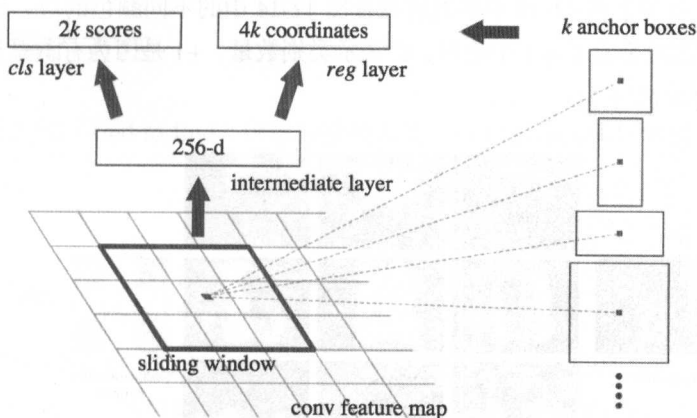


图 11-12 RPN 网络结构^[12]

Faster R-CNN 的主要步骤如下。

- (1) 特征提取：同 Fast R-CNN，以整张图片为输入，利用 CNN 得到图片的特征层。
- (2) 区域提名：在最终的卷积特征层上为每个点利用 k 个不同的矩形框（Anchor Box）进行提名， k 一般取 9。
- (3) 区域判定和回归：对每个矩形框对应的区域进行 object/non-object 二分类，并用 k 个回归模型（各自对应不同的矩形框）微调候选框位置与大小。
- (4) 分类与回归：对区域提名网络给出的区域结果进行筛选，进行目标分类和边框回归。

总之，Faster R-CNN 抛弃了选择性搜索，引入了 RPN 网络，使得区域提名、分类、回归一起共用卷积特征，从而得到进一步的加速。但是，Faster R-CNN 仍然分成两步：对两万个矩形框先判断是否是目标（目标判定），然后再进行目标识别。

11.2.5 R-FCN

前面的目标检测方法都可以细分为两个子网络：共享的全卷积网络和不共享计算的与 ROI 相关的子网络（比如全连接网络）。

R-FCN^{[14][15]} 将最后的全连接层之类的换成了一个位置敏感的卷积网络，从而让所有计算都可以共享。具体来说，先把每个提名区域划分为 $k \times k$ 个网格，比如 R-FCN 原论文中 k 的取值为 3，则对应的 9 个网格分别表示：左上（top-left）、上中（top-center）……右下（bottom-right），对应于图 11-13 中的九宫格及图 11-14 中的不同颜色的块，每个网格都有对应的编码，但预测时会有 $C+1$ 个输出， C 表示类别数量，+1 是因为有背景类别，全部的输出通道数量为 $k^2 \times (C+1)$ 。

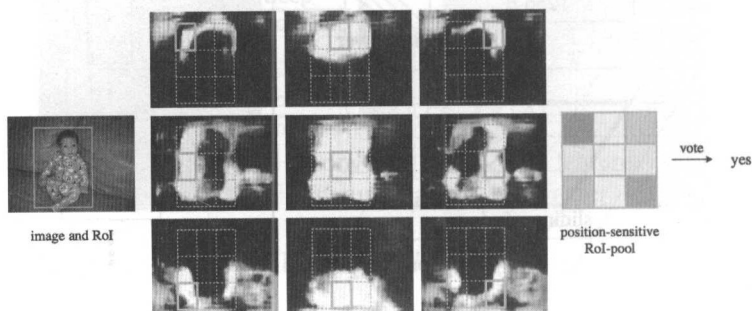


图 11-13 R-FCN 的 person 分类可视化过程

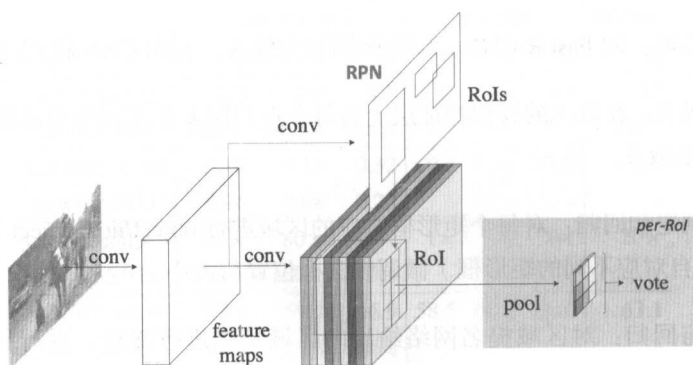


图 11-14 R-FCN 网络结构

需要注意的是，图 11-13、图 11-14 中的不同位置都存在一个九宫格，但是池化时只有一个起作用，比如 bottom-right 层只有右下角的小块起作用。那么问题来了，这一层其他的

8 个框有什么作用呢？答案是它们可以作为其他 ROI（偏左或偏上一些的 ROI）的右下角。

R-FCN 的步骤如下。

（1）区域提名：使用 RPN（Region Proposal Network，区域提名网络），RPN 本身是全卷积网络结构的。

（2）分类与回归：利用和 RPN 共享的特征进行分类。当进行 bbox 回归时，则将 C 设置为 4。

11.3 端到端的方法

本节介绍端到端（End-to-End）的目标检测方法，这些方法无需区域提名，包括 YOLO 和 SSD。

11.3.1 YOLO

YOLO^{[16][17]} 的全称是 You Only Look Once，顾名思义，就是只看一次，进一步把目标判定和目标识别合二为一，所以识别性能有了很大提升，达到每秒 45 帧，而在快速版 YOLO（Fast YOLO，卷积层更少）中，可以达到每秒 155 帧。

YOLO 网络的整体结构如图 11-15 所示，针对一张图片，YOLO 的处理步骤为：

- （1）把输入图片缩放到 448×448 大小。
- （2）运行卷积网络。
- （3）对模型置信度卡阈值，得到目标位置与类别。

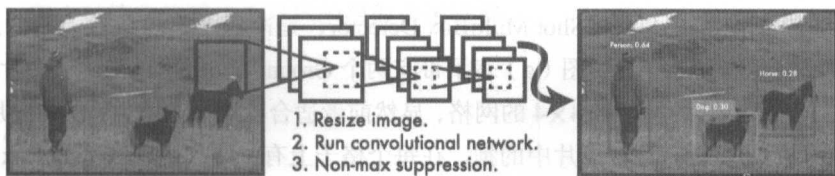


图 11-15 YOLO 网络结构

YOLO 网络的模型如图 11-16 所示，将 448×448 大小的图片切成 $S \times S$ 的网格，目标中心点所在的格子负责该目标的相关检测，每个网格预测 B 个边框及其置信度，以及 C 种类别的概率。在 YOLO 中 $S = 7$ ， $B = 2$ ， C 取决于数据集中物体类别数量，比如 VOC 数据集就是

$C = 20$ 。对 VOC 数据集来说，YOLO 就是把图片统一缩放到 448×448 ，然后每张图平均划分为 $7 \times 7 = 49$ 个小格子，每个格子预测 2 个矩形框及其置信度，以及 20 种类别的概率。较大的物体可能会由多个网格单元提名，YOLO 采用了 NMS（Non-Maximum Suppression，非最大抑制）的方法来过滤结果。NMS 将 mAP 提升了 2~3 个百分点。

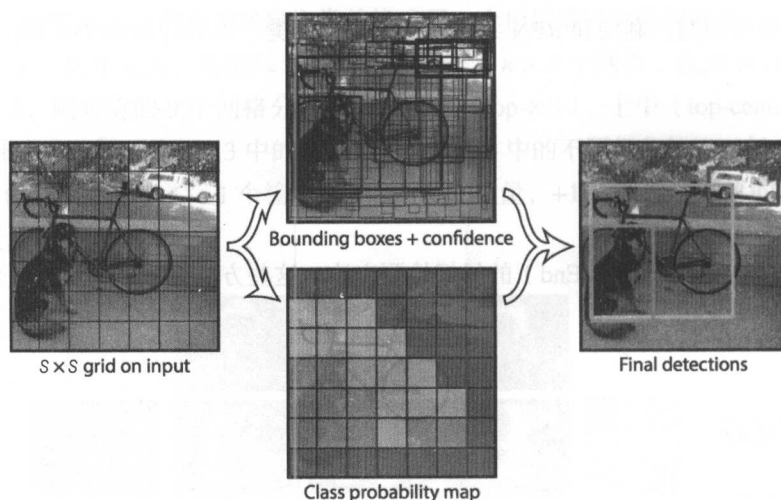


图 11-16 YOLO 网络模型

YOLO 简化了整个目标检测流程，速度的提升也很大，但是 YOLO 还是有不少可以改进的地方，比如 $S \times S$ 的网格就是一个比较启发式的策略，如果两个小目标同时落入一个格子中，模型也只能预测一个；另一个问题是损失函数对不同大小的 bbox 未做区分。

11.3.2 SSD

SSD^{[18][19]} 的全称是 Single Shot MultiBox Detector，是冲着 YOLO 的缺点来的。SSD 的基本框架如图 11-17 所示，其中图 (a) 表示带有两个 Ground Truth 边框的输入图片，图 (b) 和 (c) 分别表示 8×8 的网格和 4×4 的网格，显然前者适合检测小的目标，比如图片中的猫；后者适合检测大的目标，比如图片中的狗。在每个格子上有一系列固定大小的 Box（有点类似于前面提到的矩形框），这些在 SSD 中称为 Default Box，用来框定目标物体的位置，在训练时 Ground Truth 会赋给某个固定的 Box，比如图 (b) 中的深色框和图 (c) 中的深色框。

SSD 网络分为两部分，前面的网络是用于图像分类的标准网络（去掉了与分类相关的层）；后面的网络是用于检测的多尺度特征映射层，从而实现检测不同大小的目标。SSD 和 YOLO 的网络结构对比如图 11-18 所示。

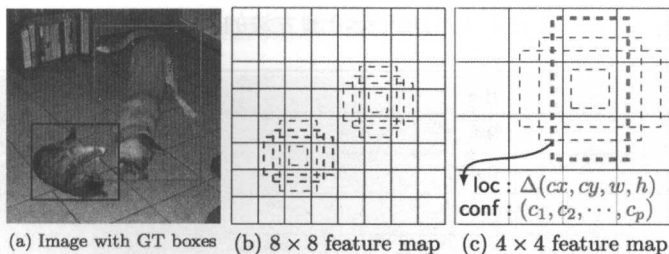


图 11-17 SSD 基本框架

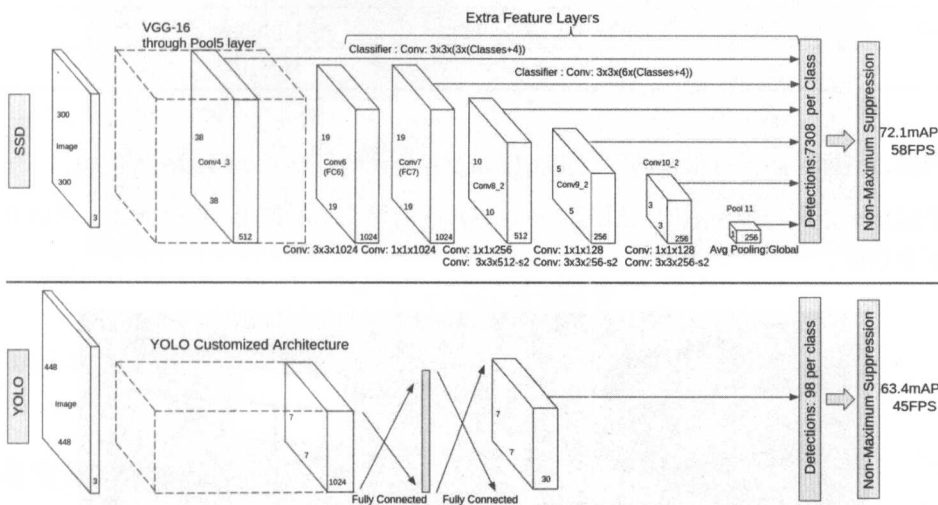


图 11-18 SSD 和 YOLO 的网络结构对比

SSD 在保持 YOLO 高速的同时效果也提升很多,主要是借鉴了 Faster R-CNN 中的 Anchor 机制,同时使用了多尺度。但是从原理依然可以看出,Default Box 的形状以及网格大小是事先固定的,那么对特定的图片小目标的提取会不够好。

11.4 小结

基于深度学习的目标检测总体上分为基于区域提名的 R-CNN 系列和无需区域提名的 YOLO、SSD 系列。

表 11-1 大致对比了各种方法的性能 (fps, 每秒帧数) 和 VOC 2007 上的 map 对比。注意: 相关数据搜集自不同的论文,由于评测硬件和环境等的区别,数据仅供参考,不具有绝对的对比意义。

表 11-1 不同目标检测方法的指标对比

方法	fps	VOC 2007
OverFeat	0.5	
R-CNN	0.077	48%~66%
SPP-net		63.1%~82.4%
Fast R-CNN		66.9%~70%
Faster R-CNN	15 (ZF Model)	73.2%~85.6%
R-FCN	6	83.6%
YOLO	45~150	58.8%
SSD	58~72	75.1%

当然，目标检测还有很长的路要走，比如业界公认的较难的小目标检测问题。

小试身手，来一张实际的三里屯照片，YOLO 的检测结果如图 11-19 所示，可以看出漏检了不少目标。



图 11-19 YOLO 检测结果

再来看看图 11-20 所示的 SSD 检测结果, 看起来效果好不少, 但被遮挡的人还是漏检了。



图 11-20 SSD 检测结果

期待未来基于深度学习的目标检测的进一步突破!

参考文献

- [1] Felzenszwalb P F, Girshick R B, McAllester D, et al. Object detection with discriminatively trained part-based models[J]. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 2010, 32(9): 1627-1645.
- [2] C. Szegedy, A. Toshev, D. Erhan. Deep Neural Networks for Object Detection. Advances in Neural Information Processing Systems 26 (NIPS), 2013.
- [3] P. Sermanet, D. Eigen, X.Zhang, M. Mathieu, R. Fergus, and Y. LeCun. OverFeat: Integrated recognition, localization and detection using convolutional networks. In ICLR, 2014.
- [4] R. Girshick, J. Donahue, T. Darrell, J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. ImageNet Large-Scale Visual Recognition Challenge workshop, ICCV, 2013.
- [5] R. Girshick, J. Donahue, T. Darrell, J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014.

- [6] R. Girshick, J. Donahue, T. Darrell, J. Malik. Region-Based Convolutional Networks for Accurate Object Detection and Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, May. 2015.
- [7] R-CNN: Region-based Convolutional Neural Networks. <https://github.com/rbgirshick/rcnn>.
- [8] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In ECCV. 2014.
- [9] SPP-net. https://github.com/ShaoqingRen/SPP_net.
- [10] Girshick, R. Fast R-CNN. ICCV 2015.
- [11] Fast R-CNN. <https://github.com/rbgirshick/fast-rcnn>.
- [12] S. Ren, K. He, R. Girshick, J. Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. Advances in Neural Information Processing Systems 28 (NIPS), 2015.
- [13] Faster R-CNN. <https://github.com/rbgirshick/py-faster-rcnn>.
- [14] R-FCN: Object Detection via Region-based Fully Convolutional Networks. Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. Conference on Neural Information Processing Systems (NIPS), 2016.
- [15] R-FCN. <https://github.com/daijifeng001/r-fcn>
- [16] Redmon, J., Divvala, S., Girshick, R., Farhadi, A.. You only look once: Unified, real-time object detection. In: CVPR. (2016).
- [17] YOLO. <http://pjreddie.com/darknet/yolo/>.
- [18] Liu W, Anguelov D, Erhan D, et al. SSD: Single Shot MultiBox Detector[J]. arXiv preprint arXiv:1512.02325, 2015.
- [19] SSD. <https://github.com/weiliu89/Caffe/tree/ssd>.
- [20] J. Hosang, R. Benenson, P. Dollár, and B. Schiele. What makes for effective detection proposals? TPAMI, 2015.
- [21] J.R. Uijlings, K.E. vandeSande, T. Gevers, and A.W. Smeulders. Selective search for object recognition. IJCV, 2013.
- [22] OverFeat source code. <http://cilvr.nyu.edu/doku.php?id=software:OverFeat:start>.
- [23] ILSVRC 2013 results. <http://www.image-net.org/challenges/LSVRC/2013/results.php>.

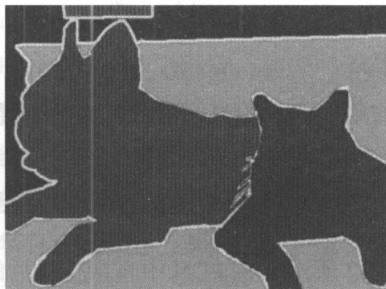
12

语义分割

图像语义分割，顾名思义，就是按照语义对每个像素点进行分类。以图 12-1 为例，语义分割问题不仅需要解决这张图片中包含哪些物体，还需要解决每个像素点属于哪个物体。如图 12-1 (b) 所示为对图 12-1 (a) 这张图片的语义分割结果。其中，左侧部分属于狗，右侧部分属于猫。



(a) 原图



(b) 语义分割结果

图 12-1 图像语义分割示例^[1]

近年来，随着深度学习尤其是 CNN 网络所取得的重大进展，图像语义分割的效果也有了非常大的进步。自从深度学习开始解决图像语义分割问题以来，已经形成了一个基本的流程：

- (1) 用全卷积神经网络来获取基准的像素分类。
- (2) 通过 CRF/MRF 来使用全局信息使得像素分类的准确率进一步提升。

大多数研究的进展都是围绕流程 1 或/和流程 2 来不断突破的。流程 1 的代表性方法有 FCN^[1], DeconvNet^[2], SegNet^[3], DilatedConvNet^[4], DeepLab^[5] 等, 其中 FCN 由于其工作的开创性获得了 CVPR'2015 的最佳论文奖; 流程 2 的代表性方法有 DeepLab^[5], CRFasRNN^[6], DeepParsing^[7]。

语义分割一般只需要解决像素的分类问题, 如果一张图片上具有同样种类的多个实例, 那么这些像素是不需要加以区分的, 即这些像素将共享同一个标签。如果需要区分每个像素究竟属于哪个实例, 那么问题更加复杂, 目前 Cascade^[8] 文章通过将图像检测和语义分割结合起来解决这个问题。

12.1 全卷积网络

12.1.1 FCN

语义分割面临着语义和位置的内在矛盾: 全局信息解答是什么, 局部信息解答在哪里。FCN 定义了一个“skip”结构来合并深层的、粗粒度的语义信息和浅层的、细粒度的位置信息。FCN 的成功是建立在深度网络在图像分类和迁移学习基础上的。FCN 使用并拓展了现有的图像分类模型, 然后通过整张图片的输入和 Ground Truth 来微调卷积层的参数。

卷积的平移不变性: 卷积层 (Convolution, ReLU, Pooling) 的操作都只与感知区内的低层特征图相关。

FCN 与用于识别的深度学习网络的主要区别如下。

1. 如何把图片分类中粗粒度的特征图还原到像素级别

如图 12-2 所示, 在 FCN 中直接把最后的全连接层用卷积层代替, 第一个卷积层对应的卷积核的 kernel 是 7×7 , 第二个卷积层对应的卷积核的 kernel 是 1×1 。这两个卷积层后都跟着 Dropout 层。全卷积网络的输出天然就很适合解决这种稠密的问题。由于网络是全卷积网络, 所以它可以使用任意尺寸的输入图片, 但是输出维度被下采样所减小。

2. 反卷积层做上采样

除全卷积网络以外, 另一个能够将粗粒度输出变为稠密输出的方法是上采样。最简单的, 比如双线性插值, 可以从最近的几个点估计填充的点的值。另外, 上采样 f 倍, 可以看作卷积操作的步长是 $\frac{1}{f}$ 。一种直观的方式是使用反卷积 (Deconvolution)^[9], 它的前向、后向传播方

式与卷积的后向、前向传播方式一致。一组反卷积和激活函数可以构造非线性的上采样。一些学者认为“Deconvolution”这个名称不好,而应该命名为“Inverse of Convolution”“Convolution Transpose”“Backward Strided Convolution”“1/2 Strided Convolution”“Upconvolution”等。参考文献 [10] 和 [1] 中使用了两种上采样的方式,在后面的章节中将会详细介绍。

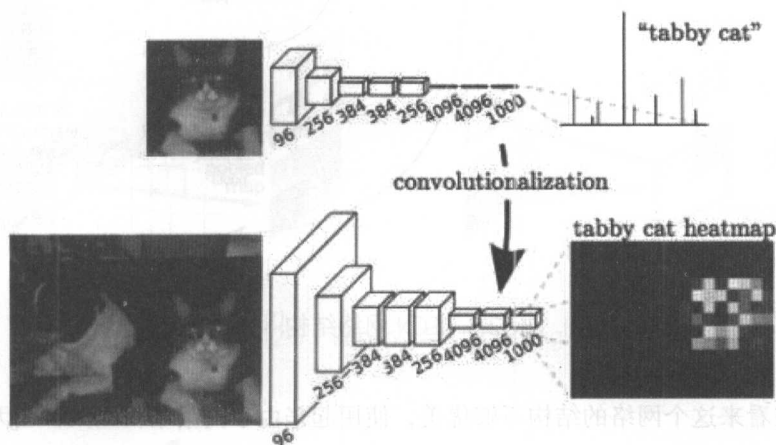


图 12-2 全卷积网络的构造^[1]

3. 切分网络结构

FCN 借用了 ILSVRC 分类网络 (AlexNet^[11], VGGNet^[12], GoogLeNet^[13]) 的参数,然后在这个网络后面增加了用于稠密计算的上采样和像素级损失函数,通过微调的方式来训练新增网络结构的参数。如图 12-3 所示,这个网络结构的一大亮点在于浅层信息和深层信息的合并,也就解决了本节开始提出的如何把全局的语义信息与局部的分割信息相融合的问题。图 12-3 中的 FCN-32s 直接在最后一个卷积层的特征图后做了上采样,这一步采用的是不可学习的双线性插值上采样。FCN-16s 和 FCN-8s 分别对最后一个卷积层的特征图做了 2 倍和 4 倍的上采样,这里的上采样的参数初始化为双线性插值,在训练过程中参数可学。实验证明,FCN-8s 的效果最好。由于这个网络的参数太多了,而用于分割的训练样本数又较少,所以作者使用其他的分类问题来训练好初始的网络,然后再微调后面的上采样过程。

FCN 的每个解码器学习上采样的参数,上采样得到特征图后,与对应的编码特征图合并,作为输入传送给下一个解码器。编码器的参数数量巨大 (134M),但是解码器的参数数量非常少 (0.5M)。由于参数太多了,端到端的训练难以直接完成,所以采用了分步的训练方式。每个解码层逐个加入已经训练好的网络中,直到性能不再提升时,就不再加入新的解码层了。

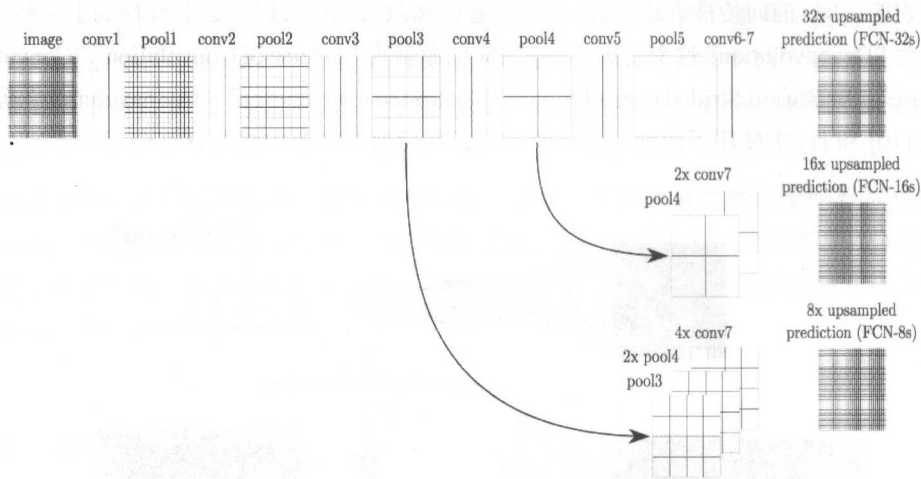


图 12-3 FCN 网络结构^[1]

尽管现在看来这个网络的结构不够优美，使用起来由于网络结构太大，也无法完成端到端的训练，由于网络固定了感受野的大小，导致无法准确预测太大的物体的边缘或者容易丢失较小的物体，但它开创了用深度卷积神经网络解决语义分割问题的先河，拿到了 CVPR'2015 最佳论文奖确实是实至名归。

12.1.2 DeconvNet

在 FCN 发表半年后的 ICCV'2015 上，DeconvNet 针对 FCN 的部分缺点做了优化，使用反卷积^[9]和 Unpooling 来弥补 FCN 规定了维度的缺点。其全局的网络结构如图 12-4 所示。

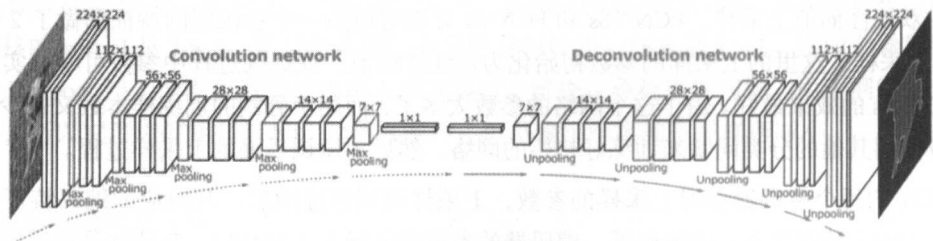


图 12-4 DeconvNet 网络结构^[2]

卷积神经网络中的池化层用于从一定的感受野中过滤噪声，提取单一特征，这在分类

问题中非常有效。但是在语义分割问题中，细节同样重要，所以 Unpooling 操作非常有效。Unpooling 与池化层相对应，池化层记录好每个结果对应的来源（下标）传递给 Unpooling 层，如图 12-5 所示。图 12-6 示意了反卷积操作，尽管看起来反卷积操作将特征图扩大了，但是 DeconvNet 中的反卷积层保持特征图不变。特征图的增大是通过 Unpooling 来实现的。

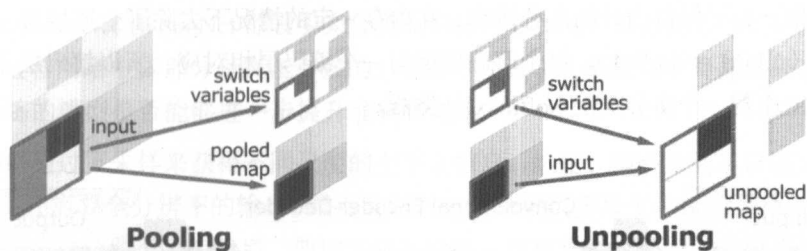


图 12-5 Pooling 与 Unpooling^[2]

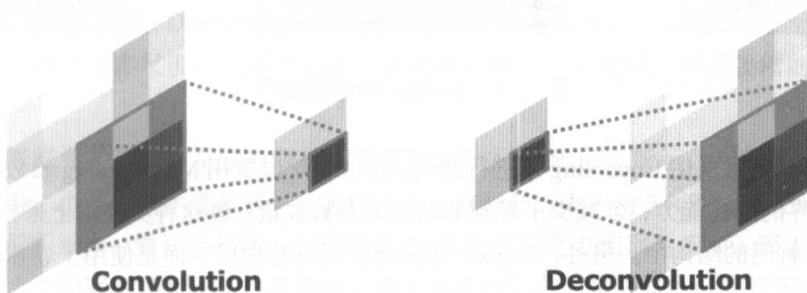


图 12-6 卷积与反卷积^[2]

与 FCN^[1] 相比，这种结构化的反卷积能够获得不同层次的形状和细节，弥补了 FCN 在特大或特小物体语义分割上的缺陷。

DeconvNet 也同样遇到了网络结构复杂、训练样本少的问题。为了解决这个问题，DeconvNet 在每个卷积层和反卷积层增加了 Batch Normalization^[14]。DeconvNet 还采用了分段训练的方式，先使用简单的样本训练网络，然后再用复杂的样本微调网络。为了获得简单的训练样本，先裁剪图片，使得目标物体位于图片中央。在推断时，采用了区域提名的技术来辅助，将多个区域的结果合并作为最终结果呈现。

12.1.3 SegNet

与 FCN 和 DeconvNet 基本同时期，剑桥大学的 Vijay Badrinarayanan 等学者提出了 SegNet^[3]，它也采用全卷积神经网络来解决图像语义分割问题。SegNet 的网络结构如图 12-7 所示。编码网络由 VGGNet16 的前 13 个卷积层构成。初始化训练参数时采用了 ImageNet 这个更大的数据集。为了保留图片的高分辨率，在内存一定的情况下去除了全连接层，参数数量大幅度下降（134M \rightarrow 14.7M）。每个解码层都与一个编码层相对应，所以解码层也有 13 层。解码层的输出连着一个多分类的 Softmax 分类器。

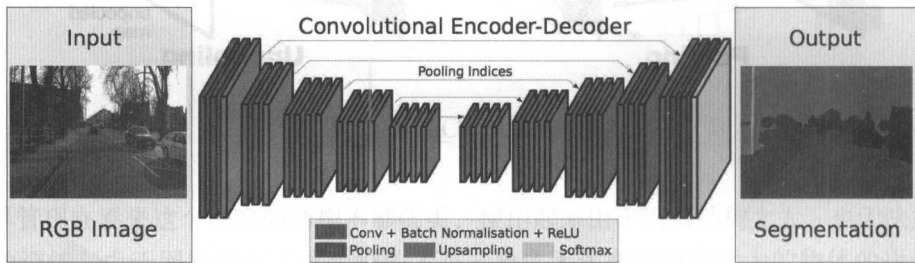


图 12-7 SegNet 网络结构^[3]

在 SegNet 中也采用 Unpooling 来进行解码。在解码阶段重用 Max-Pooling 参数的好处是：① 提升了边界的描述能力；② 减少了端到端训练的参数数量；③ 这种类型的上采样可以应用到任何编码-解码的结构中。另外，SegNet 并没有使用反卷积层，而是使用了卷积层。

与 FCN 和 DeconvNet 相比，SegNet 减少了原来分类网络中的全连接层，内存消耗和计算时间都有优势。由于参数数量的减少，采用 SegNet 比较容易实现端到端的训练。

对图像语义分割效果的判断一般采用如下几种方式。

- 全局均值 (Global Average)：按像素正确分类的比例计算，这个指标的计算方式简单，容易应用在深度神经网络中。但是它没有考虑不同种类物体的差别，它与人类的认知也有一定的差别。
- 分类平均准确率 (Class Average Accuracy)：所有分类预测准确率的平均值。
- 交集/并集的均值 (Mean Intersection Over Union, mIoU)：mIoU 比分类平均准确率更严格，因为它惩罚了伪正例 (False Positive)。但是它的缺点是，交叉熵损失不好直接优化。另外，它只是衡量了像素点正确分类的总数，并没有精确地描述切分边界的准确性。

- 语义轮廓得分 (Semantic Contour Score): 在给定容忍距离的情况下, 计算预测边界和真实边界的 F1-measure。语义轮廓得分和 mIoU 组合的方式与人类的认知最接近。

12.1.4 DilatedConvNet

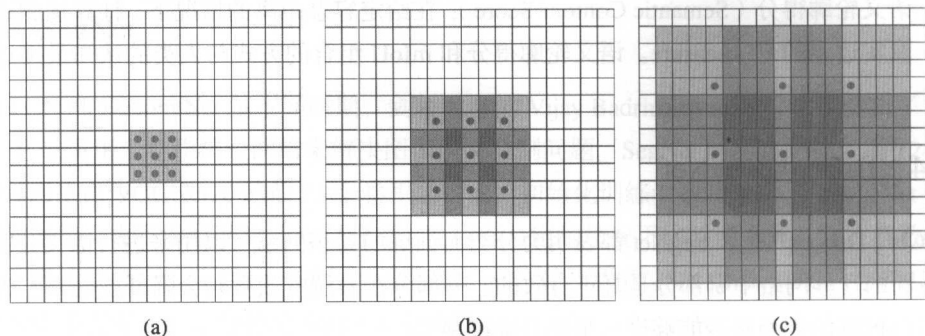
FCN、DeconvNet 和 SegNet 都表明为分类而训练的卷积网络可以用来进行语义分割。那么, 在新的迁移网络中哪部分是真正有效的? 哪部分在做稠密预测时会降低准确率? 为稠密网络专门训练的模型是否能够进一步提升准确率呢?

分类网络通过下采样来获得不同尺度的上下文信息。然而, 稠密预测不仅需要多尺度的上下文信息, 还需要全分辨率的输出。最近有两种方法来解决多尺度和全分辨率输出的矛盾: 第一, 通过下采样获取全局信息后, 使用多层上采样-卷积来恢复丢失的分辨率信息; 第二种, 使用多尺度的输入图片作为多个输入, 然后把不同输入的预测结合起来^{[15][16][17]}。

DilatedConvNet^[4] 与前面所述的三种网络结构不同, 前三种网络都是在分类网络的基础上加入了解码结构, 使得原来求解分类问题的网络变为求解稠密问题的网络。而 DilatedConvNet 则使用了膨胀卷积 (Dilated Convolution), 在不损失分辨率、不需要借助多尺度输入图片的基础上, 融合了多尺度的上下文信息。这是卷积层的一个变种, 不需要借助降采样/池化层。

那么, 膨胀卷积是如何呈指数级扩大感受野的呢? 如图 12-8 所示, 灰色区域表示原输入中的感受野, 小圆点表示卷积核对应的输入点, 卷积核的大小为 3×3 。其中图 (a) 表示的是传统的卷积操作, 卷积核与输入中连续的 3×3 的区域逐点相乘再相加。图 (b) 表示的是膨胀参数为 2 的膨胀卷积层, 也就是说, 这个卷积核要跟原输入对应的感受野中的每隔 1 个像素点的位置进行相乘再相加的操作, 在卷积核的大小仍然是 3×3 的情况下, 感受野扩大到 7×7 , 相当于增加了一个池化层。如果在图 (b) 的基础上继续膨胀, 图 (c) 表示的是膨胀参数为 4 的膨胀卷积层, 也就是说, 这个卷积核要跟原输入对应的感受野中的每隔 3 个像素点的位置进行相乘再相加的操作, 此时的感受野已经扩大到 15×15 , 相当于增加了两个池化层。

DilatedConvNet 的实现也以 VGGNet16 为蓝本, 但是去掉了所有的池化和步长 (Stride), 而是以膨胀卷积层来扩大感受野。实验指出, 去掉池化层和中间层的填充 (Padding), 会提升准确率。

图 12-8 感受野呈指数级增长的膨胀卷积^[4]

12.2 CRF/MRF 的使用

12.2.1 DeepLab

对于精准的目标分割来说，DCNN（Deep Convolutional Neural Network）的最后一层的细节不够充分，这是由 DCNN 的不变性决定的。DCNN 的不变性使它在高层任务（图片分类、目标检测）上的表现非常好。然而，把 DCNN 直接用于语义分割，需要克服两方面的缺点：一是信号降采样；二是空间不变性。降采样是由池化层和卷积层里的步长带来的，DeepLab^[5] 为了克服降采样的缺点，先于 DilatedConvNet^[4] 采用了膨胀卷积，但是它保留了池化层。膨胀卷积的细节可以参考上一节的内容。第二个问题则是由以获取目标来做决策的分类问题跟与空间信息相关的分割问题的矛盾引发的。DeepLab 为了解决第二个问题，采用了条件随机场 DenseCRF^[18]。之所以使用 DenseCRF，是由于它不仅运算速度快，而且既保留了细节，又能获得长距离的依赖关系。可以说，DeepLab 的贡献在于结合了深度卷积神经网络和概率图模型这两大机器学习方法。

DCNN 的得分映射（Score Map，Softmax 层的输入）能够预测物体的大概位置，但是边界定位得不够好。卷积层天然使得在分类准确性和定位准确性之间存在取舍：在分类网络中，具有多个池化层的更深的模型在分类问题上更准确，但是它的位置无关性和更大的感受野使得在得分映射上计算准确位置更难。

传统的 CRF 使用能量函数来建模相邻节点，使相近的像素点更倾向于相同分类。这与图像语义分割的目标不同：得分映射已经非常平滑，相近的像素点已经具有同质化的分类。在这种情况下，使用短距离的 CRF 可能会有害。语义分割的目标是还原细节，但不是进一

步平滑。所以 DeepLab 采用了长距离的 CRF，模型的能量函数如下：

$$E(x) = \sum_i \theta_i(x_i) + \sum_{ij} \theta_{ij}(x_i, x_j)$$

其中 x 是像素点的分类。一元势函数来自前端的输出：

$$\theta_i(x_i) = -\log P(x_i)$$

其中 $P(x_i)$ 是 DCNN 计算出来的在 i 像素点的分类概率。第二项是二元势函数，当 i, j 两点的分类相同时，此项取值是 0。整张图片上的任意两点间都存在二元势函数，而不论这两点距离有多远。二元势函数定义如下：

$$\omega_1 \exp\left(-\frac{\|p_i - p_j\|^2}{2\sigma_\alpha^2} - \frac{\|I_i - I_j\|^2}{2\sigma_\beta^2}\right) + \omega_2 \exp\left(-\frac{\|p_i - p_j\|^2}{2\sigma_\gamma^2}\right)$$

其中 p 表示位置， I 表示颜色。第一个核与位置和颜色相关，第二个核只与位置相关。超参数 σ_α 、 σ_β 和 σ_γ 控制了高斯核的方差。第一个高斯核使得相近颜色的相邻像素点更容易拥有同样的分类，第二个高斯核只考虑了相邻像素点之间的平滑度。图 12-9 显示了 CRF 在描绘细节方面的能力。

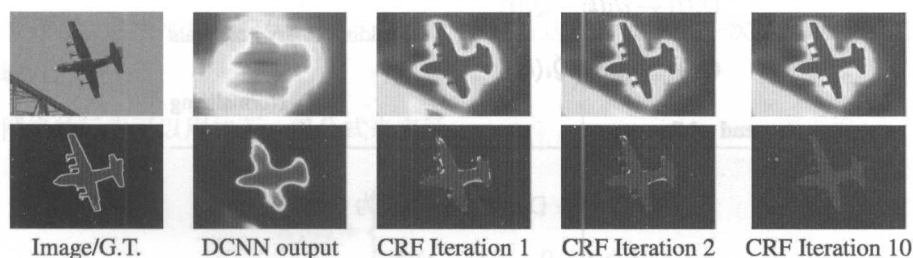


图 12-9 使用 CRF 精准分割

DeepLab 还使用了多尺度预测的方法，在输入图片和前四个池化层的输出后都增加两层感知机（第一层是 128 个 3×3 的卷积层，第二层是 128 个 1×1 的卷积层），然后与主网络的最后一层的特征图并列，所以 Softmax 层的输入增加了 $5 \times 128 = 640$ 个通道。训练时，保持主网络的旧参数不变，只更新新增的参数。

DeepLab 使用 CRF 后，提升 4%；使用多尺度特征后，提升 1.5%；既使用多尺度又使用 CRF 后，提升 5.4%。

12.2.2 CRFasRNN

DeepLab 使用 CRF 来细调语义分割结果，准确率获得了非常大的提升。然而，DeepLab 中的深度卷积神经网络和条件随机场相互独立，可以说是在 DCNN 的结果后面增加了一个独立的概率图模型。DCNN 的训练与后面 CRF 的训练相对独立，没有办法获得 CRF 的信息。由于 CRF 并没有与 DCNN 完全融合在一起，CRF 的能力还没有得到完全的发挥。而 CRFasRNN^[6] 的贡献在于把 CRF 建模成深度神经网络的一部分，实现了端到端深度学习的解决方案。首先用 CNN 来建模 CRF 的一次迭代，然后用 RNN 来建模所有迭代。

与传统的 CNN 在训练后就确定了参数的做法不同，这里的 CNN 的系数取决于图片上的原始信息，虽然高斯核的大小跟原始图片一样，但是它只需要很少的参数。CRF 的参数如高斯核的权重和标签相容性函数在整个网络的训练中得到了优化，能够进行前向、后向传播。如图 12-10 所示，参考文献 [6] 给出了将 DenseCRF 中的平均场建模为 CNN 的步骤。

```

 $Q_i(l) \leftarrow \frac{1}{Z_i} \exp(U_i(l))$  for all  $i$                                 ▷ Initialization
while not converged do
   $\tilde{Q}_i^{(m)}(l) \leftarrow \sum_{j \neq i} k^{(m)}(\mathbf{f}_i, \mathbf{f}_j) Q_j(l)$  for all  $m$ 
                                                                    ▷ Message Passing
   $\check{Q}_i(l) \leftarrow \sum_m w^{(m)} \tilde{Q}_i^{(m)}(l)$ 
                                                                    ▷ Weighting Filter Outputs
   $\hat{Q}_i(l) \leftarrow \sum_{l' \in \mathcal{L}} \mu(l, l') \check{Q}_i(l')$ 
                                                                    ▷ Compatibility Transform
   $\check{\check{Q}}_i(l) \leftarrow U_i(l) - \hat{Q}_i(l)$ 
                                                                    ▷ Adding Unary Potentials
   $Q_i \leftarrow \frac{1}{Z_i} \exp(\check{\check{Q}}_i(l))$ 
                                                                    ▷ Normalizing
end while

```

图 12-10 DenseCRF 建模为 CNN^[6]

这里使用 $U_i(l)$ 来表示前面章节中提到的负的一元势函数。也就是说：

$$U_i(l) = -\psi_u(X_i = l)$$

其中 i 表示像素点索引， l 表示所有可能的标注中的某一个。

第一步：初始化。在每个像素点上对每个标签都做 Softmax，其中 $Z_i = \sum_l \exp(U_i(l))$ 。这一步操作没有任何额外的参数，而且可以很容易地使用 CNN 中的 Softmax 来实现前向、后向传播。

第二步：信息传递。在 DenseCRF 中，信息传递的实现是在 Q 值上使用了 m 个高斯核。高斯核的系数由像素位置和 RGB 值来决定。由于 CRF 可能是全局关联的，所以每个高斯核的感受野都需要是整张图片的大小。为了使求解过程更快，这里使用了 Permutohedral lattice^[19] 实现，具体过程可以参考文献 [18]。

第三步：高斯核结果加权求和。对每个像素点的每个标签都做一次求和操作，把前一步骤中的每个高斯核的结果做加权求和。单独考虑每个像素点的每个标签，这一项可以看作有 m 个输入通道、一个输出通道的 1×1 卷积核。

第四步：相容性变换。 $\mu(l, l')$ 表示 l 和 l' 这两个标签之间的相容性。在以往的工作中，简单地采用了指示函数来实现相容性的计算。当具有相似特性的像素点被赋予不同的标签时，采用了相同的处罚。参考文献 [6] 中则放宽了这一限制，不同的标签对具有不同的处罚。例如，<"人", "自行车"> 的处罚要比 <"天空", "自行车"> 的处罚低。这一步可以看作输入通道和输出通道都是 L （标签个数）的卷积核大小为 1×1 的卷积层。这一步就是在学习标签之间的相容性函数。

第五步：增加一元势函数。这一步没有额外的参数。

第六步：标准化。采用一个 Softmax 函数实现。

在成功地将 CRF 的一次迭代建模成 CNN 网络之后，需要解决的是如何将 CRF 的多次迭代建模成 RNN 网络。如图 12-11 所示，用 f_θ 表示一次平均场迭代做的变换，其输入是图片 I ，pixel-wise 的一维概率值为 U ，前一次迭代的边缘分布估计为 Q_{in} ，输出是下一次迭代的边缘分布。多次平均场迭代可以将上述的 f_θ 进行迭代，使用上一次迭代的 Q 和原始的 U 作为输入。

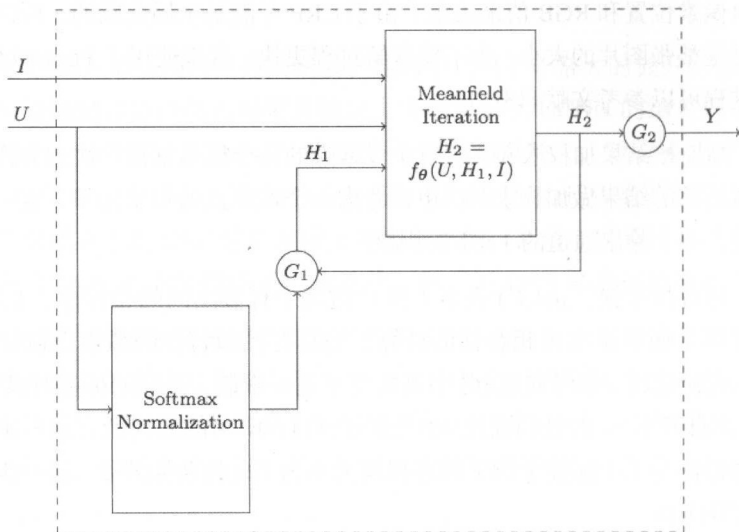
网络的行为可以用如下一组公式来表示：

$$H_1(t) = \begin{cases} \text{softmax}(U), & t = 0 \\ H_2(t-1), & 0 < t \leq T \end{cases}$$

$$H_2(t) = f_\theta(U, H_1(t), I), \quad 0 \leq t \leq T$$

$$Y(t) = \begin{cases} 0, & 0 \leq t < T \\ H_2(t), & t = T \end{cases}$$

参考文献 [18] 中提到，平均场迭代不超过 10 次就可以收敛，在实际使用中，超过 5 次迭代之后，继续迭代，就不能显著地提高效果了。所以在这个问题上，并不会遇到梯度消失或梯度爆炸的问题。因此这里采用了更简洁的 RNN 结构，并没有采用诸如 LSTM 之类的结构。

图 12-11 将 CRF 的多次迭代建模为 RNN^[6]

12.2.3 DPN

DPN (Deep Parsing Network) ^[7] 使用现有的 CNN 来完成一元问题, 然后又精心设计了其他层来模拟平均场算法的二元问题。

DPN 的优点如下:

- 在综合使用了 CNN 和随机场的一般工作中, 需要对随机场做多次迭代, 然而 DPN 只做一次迭代就可以获得比较好的效果。
- DPN 同时考虑了空间上下文关系和高阶关系, 有能力对各种二元问题进行建模, 使得很多二元问题成为 DPN 的特例。
- DPN 使得马尔可夫场问题能够并行化解决, 通过 GPU 来加速计算。DPN 使用卷积和池化操作来近似 MF, 可以通过近似来加速。

在参考文献 [5] 和 [6] 中, 二元势函数都是通过考虑任意两个像素点之间的关系来构造的。如 $\psi(y_i^u, y_j^v) = \mu(u, v) d(i, j)$ 中的 y_i^u 表示像素点 i 是否被标记为 u ; $d(i, j)$ 表示像素点 i, j 之间的关系, 比如 RGB 像素点的距离或者空间距离; $\mu(u, v)$ 则表示 u, v 这两个标记全局共现的惩罚。在空间上相近且看起来相似的两个点, 应该更容易获得相同的标签。参考文献 [7] 指

出了这种建模方式的两个缺点：①它只考虑了共现的频率，却没有考虑空间上下文关系，比如，当人和椅子一起出现的时候，人应该是坐在椅子上的，而不太可能在椅子下；②它只考虑了像素点之间成对（Pairwise）的关系，却没有考虑更高阶的关系。为了解决这些缺点，参考文献 [7] 在二元势函数中引入了三元惩罚项：

$$\Psi(y_i^u, y_j^v) = \sum_{k=1}^K \lambda_k \mu_k(i, u, j, v) \sum_{z \in \mathcal{N}_j} d(j, z) p_z^v$$

这个式子学习了局部标记上下文的混合。 K 是混合组件的数量， λ_k 取值为 0 或 1，表示哪个组件被激活，且 $\sum_{k=1}^K \lambda_k = 1$ 。如图 12-12 (b) 所示，深灰色点和浅灰色点描述了中心点 i 和它的邻域 j 。 (i, u) 表示像素点 i 被标记为 u 。 $\mu(i, u, j, v)$ 表示根据 i 和 j 的相对关系， (i, u) 和 (j, v) 同时存在的代价。上式的第二项就为三元惩罚项，表示像素点 i, j 以及 j 的邻域的关系。当 (i, u) 和 (j, v) 相容时， (i, u) 也应该相容于 (z, v) ，其中 z 是 j 的邻域，如图 12-12 (a) 所示。所以 DPN 的主要贡献就是把上式分两步建模成 CNN。第一步如图 12-12 (c) 所示，用 $m \times m$ 的卷积核作用于每个点 j 来表示 $d(j, z) q_j^v$ ，平滑了像素点 j 和它的邻域之间的预测。第二步使用 $n \times n$ 的卷积核作用于每个点 i 来表示 $\mu_k(i, u, j, v)$ ，如图 12-12 (d) 所示。

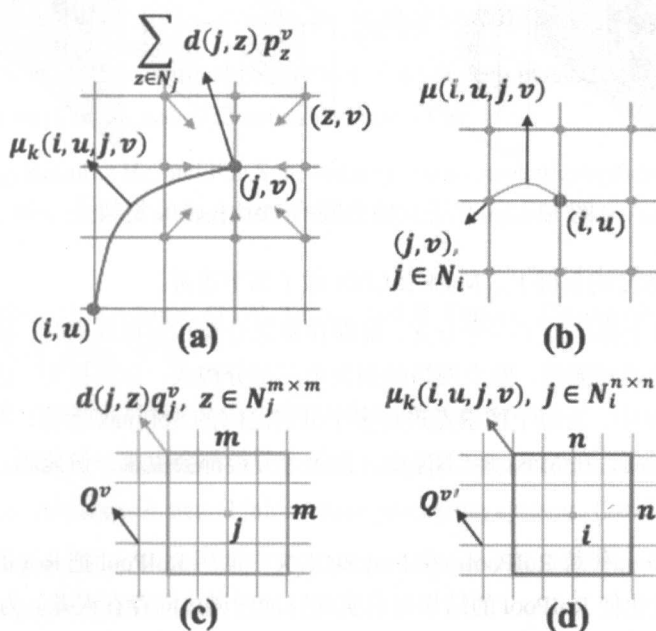


图 12-12 扩展的二元关系^[7]

12.3 实例分割

在现有的关于语义分割的研究中,实例分割的论文较少,参考文献[8]通过将目标检测的 Faster R-CNN^[20] 与 FCN 方法相结合,实现了关于实例的分割。关于 Faster R-CNN,可以通过阅读“目标检测”一章来获得更详细的介绍;关于参考文献[8],则在“多任务学习”一章中有详细的介绍,这里不再赘述。

12.3.1 Mask R-CNN

Mask R-CNN^[21] 在 Faster R-CNN^[20] 的基础上,额外增加了一个分支来预测目标的掩码。Mask R-CNN 的框架如图 12-13 所示。

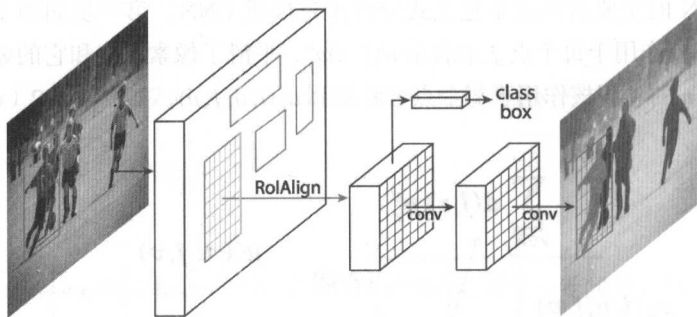


图 12-13 用于实例分割的 Mask R-CNN 框架^[21]

在 Faster R-CNN 的基础上,Mask R-CNN 做了如下改进。

- 在每个 RoI 上都增加了一个分支,能够在预测分类和边框的同时,使用一个很小的全卷积网络来得到掩码。整个网络的损失由三部分构成:分类损失、边框损失和掩码损失。在 FCN 中,为每个像素点进行多个分类之间的 Softmax 运算,不同类之间存在竞争关系。然而,在 Mask R-CNN 中,为每个分类都会生成一份掩码,不同类之间没有竞争关系。
- 使用 RoIAlign 代替 RoIPool。在 Fast R-CNN 中使用 RoIPool 把 RoI 的浮点数量化为整数,这个量化使 RoIPool 的结果与真实的特征图谱之间存在误差。为了解决这个问题, RoIAlign 使用了双线性插值来表示精确的 RoI。
- 使用了 FPN (Feature Pyramid Network)^[22] 来获取不同维度的特征。

参考文献

- [1] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. in CVPR, pp. 3431-3440, 2015.
- [2] H. Noh, S. Hong, and B. Han. Learning deconvolution network for semantic segmentation. in ICCV, pp. 1520-1528, 2015.
- [3] Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. arXiv preprint arXiv:1511.00561 (2015).
- [4] Fisher Yu, Vladlen Koltun, Multi-Scale Context Aggregation by Dilated convolutions. in ICLR 2016.
- [5] Chen, Liang-Chieh, Papandreou, George, Kokkinos, Iasonas, Murphy, Kevin, and Yuille, Alan L. Semantic image segmentation with deep convolutional nets and fully connected CRFs. In ICLR.
- [6] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. H. Torr. Conditional random fields as recurrent neural networks. in Proceedings of the IEEE International Conference on Computer Vision, pp. 1529-1537, 2015.
- [7] Liu, Ziwei, et al. Semantic image segmentation via deep parsing network. Proceedings of the IEEE International Conference on Computer Vision. 2015.
- [8] Dai, Jifeng, Kaiming He, and Jian Sun. Instance-aware semantic segmentation via multi-task network cascades. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.
- [9] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus. Deconvolutional networks. in CVPR, pp. 2528-2535, IEEE, 2010.
- [10] http://cs231n.stanford.edu/slides/winter1516_lecture13.pdf.
- [11] A. Krizhevsky, I. Sutskever, G.E. Hinton. Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. (2012) 1097-1105.
- [12] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.
- [13] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., & Anguelov, D., et al. (2014). Going deeper with convolutions. Computer Vision and Pattern Recognition (pp.1-9). IEEE.

- [14] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015.
- [15] Farabet, Clément, Couprie, Camille, Najman, Laurent, and LeCun, Yann. Learning hierarchical features for scene labeling. PAMI, 35(8), 2013.
- [16] Lin, Guosheng, Shen, Chunhua, Reid, Ian, and van den Hengel, Anton. Efficient piecewise training of deep structured models for semantic segmentation. arXiv:1504.01013, 2015.
- [17] Chen, Liang-Chieh, Yang, Yi, Wang, Jiang, Xu, Wei, and Yuille, Alan L. Attention to scale: Scale-aware semantic image segmentation. arXiv:1511.03339.
- [18] Krähenbühl, P. and Koltun, V. Efficient inference in fully connected crfs with gaussian edge potentials. In NIPS, 2011.
- [19] A. Adams, J. Baek, and M. A. Davis. Fast high- dimensional filtering using the permutohedral lattice. Computer Graphics Forum, 29(2):753-762, 2010.
- [20] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In NIPS, 2015.
- [21] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick. Mask R-CNN. <https://arxiv.org/abs/1703.06870>.
- [22] Lin, T. Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2016). Feature pyramid networks for object detection.

图像检索的深度哈希编码

图像检索，就是指给定一张待搜索图片，从数据库里面检索出相似图片的过程。为了满足实时性的搜索要求，一般会提取出图片特征并以 0、1 的哈希编码来压缩代替，这样就可以进行高效的检索了。随着互联网相关技术的发展，数据库的容量也越来越大，这在检索时就要求对哈希编码的位数加以控制。常用的位数选择为 24 位、48 位和 64 位。因此，如何得到有效的哈希编码，就成了图像检索中的关键点。

首先，我们简单看一下传统的哈希编码方法，然后介绍如何应用深度学习来进行哈希编码，以及它所带来的优势。

13.1 传统哈希编码方法

哈希编码就如同从一组图像中学习到一个映射 f ， f 可以使图像映射至二值的哈希编码上。可以想见，这是一个困难的问题。对于这个问题，传统方法的解决思路是：先对原图像用人工提取的特征来表达，即先从原图像压缩至特征表达，然后再从特征表达映射至哈希编码。

以 KSH (Supervised Hashing with Kernel) [1] 为例，其所代表的算法如下。

(1) 提取特征。

在训练集 T 内共含 M 张图，遍历 $m = 1, 2, \dots, M$ ，对第 m 张图提取人工特征，如 HoG、SIFT 等，然后将特征连接起来形成一个长向量来表达该图：feature(m)。

(2) 分解相似度矩阵。

- 将 T 按照标准的两两相似/不相似结果构建矩阵 S 。(即 1 为相似、0 为不相似, 则 S 为 0 和 1 组成的矩阵, 且 S_{ij} 为图 i 和图 j 相似与否的标注结果。)
- 将 S 分解成 HH^T , 使得 $\min |HH^T - S|$, 则 H 为训练集 T 对应的哈希编码矩阵, 其第 m 行对应图片 m 的哈希编码值。

(3) 求得特征表达至哈希值的映射 $g: \text{feature}(m) \rightarrow H(m)$ 。训练学习结束。

(4) 测试时, 对应图片 I , 先同样提取特征表达 $\text{feature}(I)$, 再根据映射 g 得到其哈希编码 $H(I) = g(\text{feature}(I))$ 。

从以上步骤可以看出, 难点集中在第 2 步, 即将相似度矩阵 S 分解成可能的哈希编码矩阵 HH^T , 并且约束 H 的取值为 0 或 1。因此, 研究者针对如何近似分解相似度矩阵做了大量艰苦的工作, 在此不再赘述。尽管如此, 传统方法还是有很大的局限性的。比如, 如果训练集过大, 则意味着 S 也会过大, 分解起来将会变得十分困难, 甚至不可达成。

而深度学习介入之后, 这些局限被逐渐打破。深度学习的应用也不是一蹴而就的, 在逐步尝试和摸索后, 终于找到了一条简捷有力的途径。

13.2 CNNH

在深度学习进行哈希编码压缩的第一篇代表性文章^[2]中提出了 CNNH (CNN-Hashing) 方法。顾名思义, 此方法应用了 CNN 卷积神经网络来优化哈希编码。方法流程如图 13-1 所示^[2], 虽然看起来很复杂, 但实际思路却非常简单, 即用 CNN 代替传统方法来进行特征提取, 以及将特征表达映射至哈希编码上。

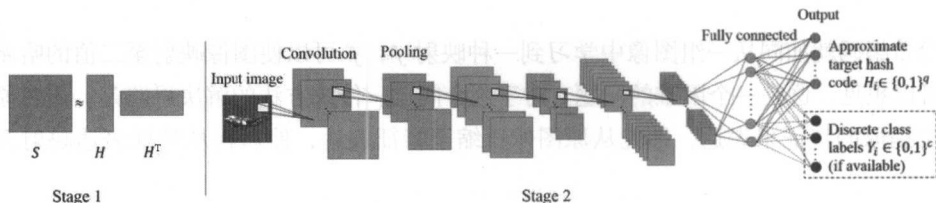


图 13-1 CNNH 的方法流程^[2]

具体来看, CNNH 的方法由两大步骤组成。其中 Stage 1 是将相似度矩阵拆解成哈希编码矩阵及其转置的乘积。这一步与传统方法的第 2 步完全相同。不同的是, 在 Stage 2 利用

CNN 一并完成了传统方法的第 1、3 步。通过前面章节的介绍我们了解到, CNN 可以完成从输入图像至输出类别的映射过程。而其中的最后几个卷积层、全连接层都分别对应一组高度抽象后的特征表达。在图 13-1 中, 输出类别对应的是最右侧的黑色的点。而在图像的哈希编码问题上, 我们也可以同样建模, 即输入为图片, 输出为在 Stage 1 已得到的哈希编码矩阵 H 。整个 Stage 2 完成了从图像提取特征表达, 直至映射到哈希编码的过程。其中哈希编码用图中最右侧的深灰色点表示。

由此可知, 此方法对传统方法的改进在于利用深度神经网络更有效地提取特征并加以映射。但另一方面, 相似度矩阵分解依然是问题的瓶颈; 而且分解后的哈希编码在 Stage 2 无法继续改进, 整个流程仍然不是端到端的简洁模型。

因此, 随后的深度哈希研究更加大胆, 直接抛弃了传统方法的束缚, 得出一个由损失函数控制、由图像到哈希编码的端到端的训练模型^{[3][4][5]}的方案。在此, 我们以 DSH^[3] 为例, 详细研究一下这个方案是如何实现的。

13.3 DSH

DSH (Deep Supervised Hashing) 是 2016 年提出的方法^[3], 它的网络结构及方法流程如图 13-2 所示。

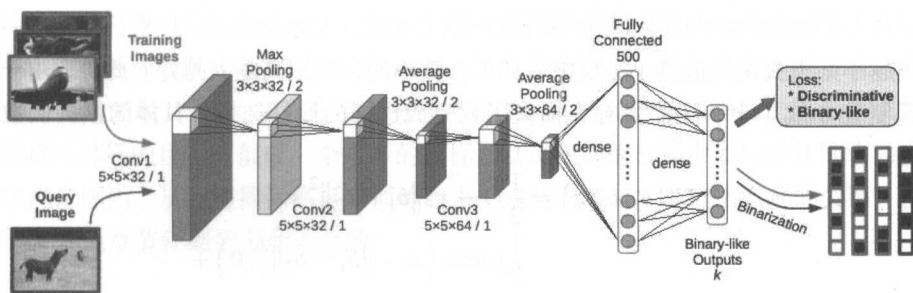


图 13-2 DSH 的网络结构及方法流程^[3]

由图 13-2 可以看出, DSH 方法不再包含分解相似度矩阵的步骤, 而是从图片到二值哈希编码一气呵成。它的网络结构为三层卷积网络接两个全连接层, 最后的全连接层就是哈希编码了。每层的具体结构在图中已经标识。

那么, 在图 13-2 与图 13-1 中的 Stage 2 无太大区别的情况下, 是如何省掉了 Stage 1 分解相似度矩阵这一步的呢?

首先, 在图 13-1 中 Stage 2 是单路模型, 即输入为一张图片, 对应的是该图片的哈希编码。然而在图 13-2 中 DSH 方法其实是双路模型, 即输入为一对图片, 输出为这一对图片的距离, 对应于其相似、不相似的标签, 如图 13-3 所示。这种双路模型也被称为 Siamese 模型, 源于 LeCun 最早的一篇双路网络模型的文献^[6]。因此, DSH 是按图片对及它们的相似标签进行训练的, 两路网络的参数共享, 因此它们是一模一样的。在测试时, 待检索的图片只需经过单路模型, 得到其哈希编码。

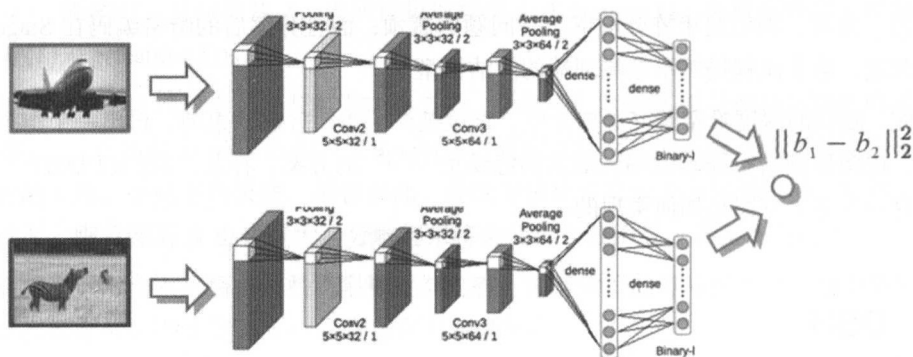


图 13-3 DSH 的双路模型^[6]

因此, 图片对的当前哈希编码距离是在最后一层的基础上得到的, 网络训练的损失函数则围绕这个距离的大小与实际相似与否的标签来进行。大体思路是, 如果标签相同, 哈希编码距离越小则损失函数值越小; 反之, 如果标签不同, 则距离越大越好。此外, 由于哈希编码要限制在 0 或 1 上, 还需要补充损失函数完成这样的偏好选择。具体的损失函数表达式如下:

$$L_r(b_1, b_2, y) = \frac{1}{2}(1-y)\|b_1 - b_2\|_2^2 + \frac{1}{2}y \max(m - \|b_1 - b_2\|_2^2, 0) + \alpha(\|b_1 - 1\|_1 + \|b_2 - 1\|_1)$$

其中 y 为标注结果, $y=0$ 表示图片相同, $y=1$ 表示图片不同。所以该损失函数表达式的第一部分对应的是在标注结果“相同”的情况下, 图片的哈希编码距离的损失以欧式距离来表示, 即距离越小损失函数值越小; 第二部分是在标注结果“不同”的情况下, 以折页损失函数 (Hinge Loss Function) 来表示, 即距离小于 m 时, 距离越小则损失函数值越大, 而当距离大于阈值 m 后, 认为已经分开得足够远了, 损失降为 0; 第三部分则为与“二值化”相关的损失, 在 DSH 中, 将哈希值的二值规定为 -1、1, 因此, 如果哈希编码的各个位的值

的绝对值越接近于 1, 则损失函数值越小。第三部分以一个权重和前边的损失结合起来, 形成总体的损失函数。

在 DSH 的训练过程中, 随着不断降低损失函数值, 不仅实现了相同、不同图像的哈希距离的优化, 而且还使得其编码值靠近二值 (-1 、 $+1$), 从而简捷地完成了端到端的哈希编码。此方法不仅绕开了传统方法在分解相似度矩阵时所面临的困难和限制, 而且由于它可以充分利用大规模的训练数据, 使得其结果比传统方法反而提高很多。这也显示了深度学习应用于此领域的成功。

13.4 小结

针对大规模数据集的快速检索要求, 将图片映射为仅几十位的哈希编码, 然后通过编码的距离计算进行图片检索。在传统方法中, 采取了一种聪明的由相似度矩阵分解生成目标哈希编码, 然后从图片到特征表达再到此目标哈希编码的映射过程。在深度学习介入之后, 不仅避开了一些复杂且成为瓶颈的步骤, 而且采用端到端简化了流程, 同时提升了效果。

从问题的提出, 到传统方法的解决策略, 再到深度学习的应用, 似乎显示了深度学习解决以往各种复杂问题轻而易举: 只需要设计合适的损失函数, 在大规模数据的帮助下, 就可以训练出不断降低该损失函数值, 从而解决问题的网络。然而, 事实上, 从概念的提出, 到最终的实现, 之间还有很多重要的路要走。比如在提出 Siamese 网络模型之前, 就有用两路模型解决这种图片对比问题的方法, 但是由于其损失函数的第二部分设计不得当 (当时未设立阈值 m , 使得不同的图片距离越远带来的损失降低没有限制, 导致模型倾向于将所有图片的距离都分得越来越开), 而使得模型没有稳定收敛。又比如哈希编码的二值化要求如何保证; 如何将不可导的目标近似成一个可导的目标, 以及在训练过程中如何保证网络的参数一直在健康的范围内, 从而使得网络可以训练出来。这些内容, 才是方法实现的关键, 也是需要读者结合其他章节仔细学习和体会的。

参考文献

- [1] S. F. Chang, Y. G. Jiang, R. Ji, J. Wang, & W. Liu. Supervised hashing with kernels. IEEE Conference on Computer Vision and Pattern Recognition, 2012.
- [2] R. Xia, Y. Pan, H. Lai, et al. Supervised Hashing for Image Retrieval via Image Representation Learning. AAAI. 2014.

[3] H. Lai, Y. Pan, Y. Liu, & S. Yan. Simultaneous feature learning and hash coding with deep neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015.

[4] H. Liu, R. Wang, S. Shan, & X. Chen. Deep Supervised Hashing for Fast Image Retrieval. IEEE Conference on Computer Vision and Pattern Recognition, 2016.

[5] F. Zhao, Y. Huang, L. Wang, et al. Deep semantic ranking based hashing for multi-label image retrieval. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015.

[6] S. Chopra, R. Hadsell, Y. Lecun. Learning a Similarity Metric Discriminatively, with Application to Face Verification. IEEE Conference on Computer Vision and Pattern Recognition, 2005.

第 3 部分

语音识别篇

14

传统语音识别基础

14.1 语音识别简介

语音识别就是把语音波形图转化成文字序列的过程。常见的机器学习任务是假设样本是独立同分布的，给定固定长度的样本特征，然后选择合适的机器学习模型做预测。然而，语音识别任务是一种时间序列标注的过程，语音长度不固定，对应输出的文字序列长度也是未知的，而且连续时间点上的语音数据不具有独立同分布的特性，增加了数据建模和模型推断的难度，即使朗读相同的文本，不同的说话人、不同的背景环境、不同的语气、不同的语速所产生的语音波形图，在长度、形状、数值上也不尽相同。语音识别主要有如下 4 个难点。

(1) 语音与文字没有一一对应的关系，不同的文字可能有相同的发音，比如英文 **night** 和 **knight**。而且很多文字的发音非常相似，这就使得基于检索的方法在语音识别里面行不通。

(2) 文字在语音里面没有清晰的边界，因为连续单词之间可能有连读，而且说话人的语速不尽相同、受环境噪音的影响，所以那种先确定单词边界，然后再做传统静态任务识别的方案也是不可行的。

(3) 连续时间点上的语音数据不具有独立同分布的特性，为其建模难度更大。

(4) 语音识别的流水线特别长，包含了声学模型打分、发音字典约束、语言模型打分和 Beam Search 贪心搜索，如图 14-1 所示。

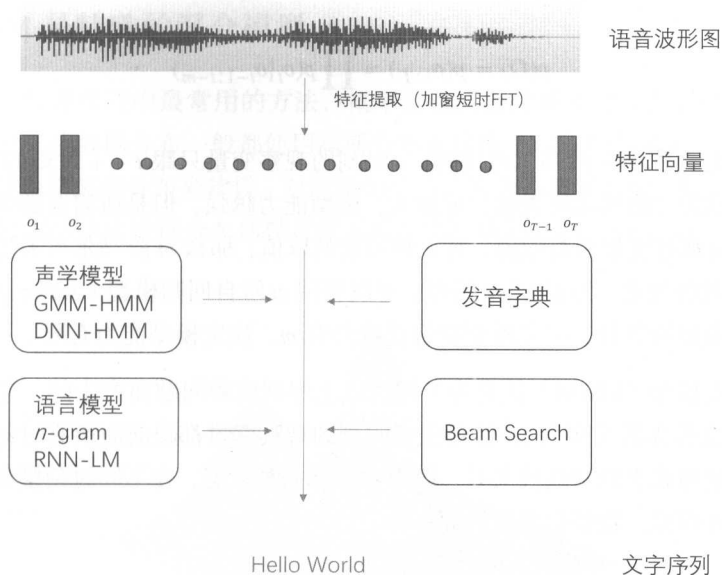


图 14-1 语音识别流程图

14.2 HMM 简介

一般我们最早接触的机器学习任务是基于数据独立同分布 (i.i.d) 假设的, 整体的数据似然就是各个数据似然的乘积, 使用最大似然估计, 取对数连乘就变成了连加, 使用基于梯度的方法可以快速优化。

$$p(O) = p(o_{1:T}) = \prod_{t=1}^T p(o_t) \quad (14.1)$$

但是在很多场景中数据独立同分布假设都是不符合实际的。例如天气预报, 今天的天气在很大程度上跟过去的天气相关; 在线手写识别, 当前笔的坐标跟之前的笔迹有很大关联; 语音识别, 当前的语音波形跟之前的语音有很大关系; 数据的联合概率可以根据乘法规则分解如下:

$$p(O) = p(o_{1:T}) = \prod_t p(o_t | o_{1:t-1}) \quad (14.2)$$

要直接为上述联合概率分布建模非常困难: 随着时间 t 的增加, o_t 依赖的数据越来越多, 模型参数越来越多, 需要的训练数据更多, 而且很容易出现过拟合。一种折中的方案是, 假设 o_t 只依赖于前 m 个观察变量, 那么似然就变成:

$$p(O) = p(o_{1:T}) = \prod_t^T p(o_t | o_{t-1:t-m}) \quad (14.3)$$

当 $m = 1$ 时, 称为一阶马尔可夫链, t 时刻的观察变量只跟 $t - 1$ 时刻的观察变量相关; 当 $m = 2$ 时, 称为二阶马尔可夫链, m 越大, 模型能力越强, 但是所需要的参数、训练数据越来越多。假设观察变量为离散值, 有 k 种可能的取值, 那么 m 阶马尔可夫链所需要的参数规模与 m 呈指数级关系, 为 $O(k^m)$ 。当然, 可以利用 m 阶自回归模型 (Autoregressive Model) 来建模以减少参数的数量, 但是模型视野长度只有 m , 注定模型能力有限。

隐马尔可夫模型 (HMM) 就是为了解决以上序列依赖问题而产生的, 它通过引入离散的隐变量 X (也称为状态序列), 使得任意时刻的观察变量都跟前面所有时刻相关, 而且所需要的参数数量与状态数呈线性关系, 因为对于同一种状态, 在不同时刻模型参数都是共享的。使用 HMM 模型, 数据似然建模如下:

$$\begin{aligned} p(O) &= \sum_X p(X, O) \\ &= \sum_X p(o_{1:N}, x_{1:N}) \\ &= \sum_X p(x_1) \left[\prod_{t=2}^T p(x_t | x_{t-1}) \right] \prod_{t=1}^T p(o_t | x_t) \end{aligned} \quad (14.4)$$

HMM 的概率图如图 14-2 所示。使用 d-separation 准则^[1], 可以看出任意两个观察变量 o_i, o_j 都有一条非阻塞的路径连接起来, 这说明任意 t 时刻的观察变量 o_t 的概率分布跟前面任意时刻的观察值 $o_{1:t-1}$ 相关。但是一旦给定状态 x_t , 当前观察值 o_t 就跟前面任意时刻的观察值 $o_{1:t-1}$ 无关了, 即 $p(o_t | x_t, o_{1:t-1}) = p(o_t | x_t)$ 。这个条件独立的性质非常重要, 使得在 HMM 计算边缘概率分布时可以使用动态规划求解 (从概率角度来理解就是利用条件独立性质对求和符号重排序)。适合使用动态规划求解的问题需要满足最优子结构条件, 该条件在概率积分里面就是指条件独立性质。

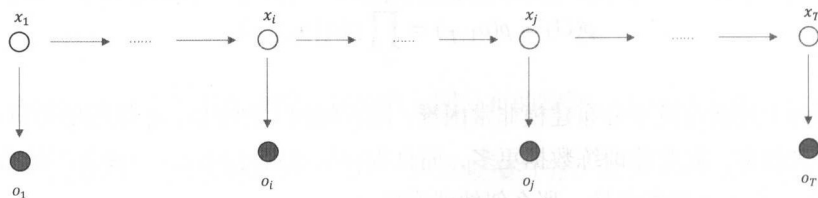


图 14-2 HMM 概率图

14.2.1 HMM 是特殊的混合模型

最大似然是机器学习中最常用的方法，概率分布或条件概率分布都需要一种分布来建模，对于简单的连续数据分布一般都使用高斯分布来建模，对应的似然函数称为高斯似然；离散数据分布使用伯努利分布来建模，对应的似然函数称为伯努利似然。但是对于复杂的数据分布场景，使用简单的数据分布建模可能不符合实际情况，例如图 14-3 所示的多峰数据分布。

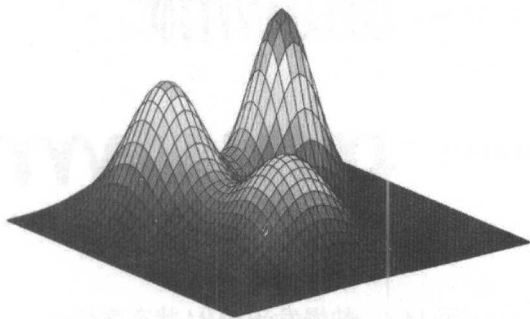


图 14-3 多峰数据分布^[13]

混合模型就是利用简单的数据分布，通过引入隐变量来构建复杂数据分布的方法。

$$p(o) = \sum_x p(o|x)p(x) \quad (14.5)$$

其中 x 为隐变量，如果 $p(o|x)$ 为高斯分布，通过引入隐变量可以得到混合高斯概率分布 $p(o)$ 。从理论上说，混合模型可以拟合任意复杂的概率分布，但是并不是说混合模型就是万能的或者高效的。建模同样复杂的数据分布，混合模型可能需要混合组件数目非常多，导致模型非常复杂，容易过拟合，而使用其他的复杂模型（如 DNN）建模可能更加简单、有效。

从式 (14.4) 可以看出，HMM 本质上也是一个混合模型，其状态就是混合模型中的隐变量，序列样本中的任意时刻的观察数据都使用混合模型来建模，稍微有点不同的地方是 HMM 多了一个隐变量之间的转移概率，这是为时序依赖关系建模而添加的，观察数据随时间的变化体现在状态序列随时间的变化上。

真实的序列数据有长有短，拿语音识别来举例，针对不同长度的文本，说出来的语音时间长度不一样，即使是相同长度的文本，不同的说话人、不同的心情、不同的语气，说出来的语音时间长度也不一样。HMM 同一状态可以在不同时刻重复出现，因此可以通过有限的离散状态取值来为任意长度的序列数据建模。HMM 本质上是一个状态机，可以把它当成一

根可以任意伸缩的弹簧，如图 14-4 所示。对于短一点的序列数据，轻微拉伸弹簧就可以撑满；对于长一点的时序数据，只要拉伸弹簧足够用力也可以撑满。总之，通过拉伸这根弹簧可以撑满任意长度的时序数据。

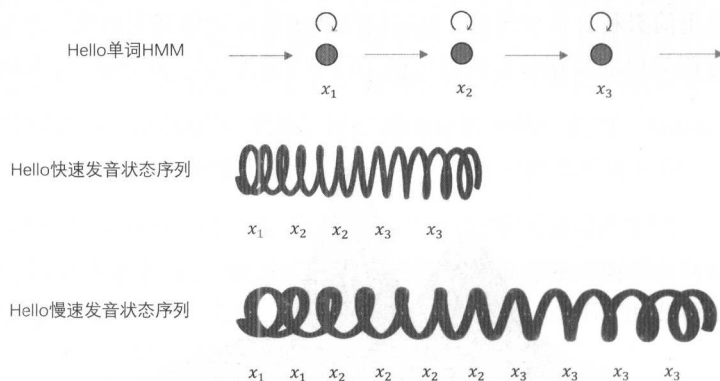


图 14-4 快慢发音 HMM 状态序列

14.2.2 转移概率矩阵

混合模型中的隐变量 x 是离散值，采用 one-hot 编码时， x 是一个 K 维向量（其中 K 是混合因子数量）。对于任意 x 向量，其中只有一个值为 1，其他值都为 0，因此其分布可以使用多项式分布来建模：

$$p(x|\pi) = \prod_{j=1}^K \pi_j^{x_j} \quad (14.6)$$

所以，HMM 初始时刻状态可以写成 $p(x_1|\pi) = \prod_{j=1}^K \pi_j^{x_{1,j}}$ 。

HMM 中的状态转移概率也是离散值，状态 i 到状态 j 的转移概率定义为 $a_{ij} = p(x_{t,j} = 1|x_{t-1,i} = 1, A)$ ，且必须满足 $\sum_j a_{ij} = 1$ 。同样采用 one-hot 编码，状态转移的多项式分布可以表示为：

$$p(x_t|x_{t-1}, A) = \prod_{i,j} a_{ij}^{x_{t-1,i} \cdot x_{t,j}} \quad (14.7)$$

状态转移示意图如图 14-5 所示。

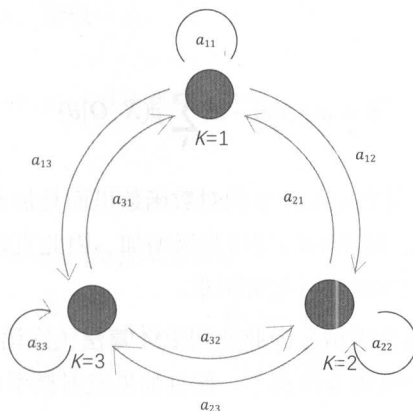


图 14-5 状态转移示意图

14.2.3 发射概率

HMM 模型独立于发射概率 $p(o_t|x_t, \Phi)$ 的选择, 其中 Φ 为发射概率模型参数。发射概率可以是离散概率值、高斯、混合高斯等概率分布, 其中 GMM-HMM 是 20 世纪 80—90 年代传统语音识别中使用最广泛的模型, 14.4.3 节将进行详细介绍。

当然, 也可以利用贝叶斯公式间接地使用神经网络等判别模型, $p(o|x) = \frac{p(x|o)p(o)}{p(x)}$, 其中的观察数据先验概率 $p(o)$ 跟模型训练和解码无关, 状态的边缘概率 $p(x)$ 可以直接从训练数据中统计出来 (通过 GMM-HMM 模型进行强制对齐得到最优状态序列, 就可以知道每一帧属于哪种状态)。给定观察数据, 状态的后验概率 $p(x|o)$ 可以使用 DNN 来建模, HMM 结合 DNN 的双向模型一般称为 DNN-HMM。它引入了深度神经网络, 相比 GMM-HMM 模型效果有很大的提升。

同样, 由于 x_t 采用 one-hot 编码, 因此发射概率可以写成:

$$p(o_t|x_t, \Phi) = \prod_{j=1}^K p(o_t|\phi_j)^{x_{t,j}} \quad (14.8)$$

14.2.4 Baum-Welch 算法

HMM 模型能很好地为序列数据建模, 模型参数包含状态初始概率 π , 转移概率矩阵 \mathbf{A} 和发射概率模型参数 Φ , 可以直接使用最大似然估计 (ML) 来进行参数求解, 等价于最小

化负对数似然：

$$\theta = \arg \min_{\theta} - \ln \sum_X p(X, O | \theta) \quad (14.9)$$

其中， $\theta = \{\pi, \mathbf{A}, \Phi\}$ 。但是式 (14.9) 中的对数函数里面是加和的形式，所有的状态序列 $X = x_1, \dots, x_T$ 有 K^T 种可能，随着时间呈指数级增加。因此直接展开上面的公式会变得异常复杂，使得基于梯度求解的方法变得非常困难。

HMM 本质上也是一个混合模型，因此使用 EM 算法（在语音识别里面叫 Baum-Welch 算法）进行参数估计会使问题变得非常简单，而且如果发射概率是高斯、混合高斯的话，那么 GMM-HMM 就是一个两层的混合模型，EM 算法能够直接得到闭式解。由于 EM 算法每次迭代时都会使得 loss 下降，所以参数求解会很高效。

E-Step: 使用目前已经求解到的参数 θ^{old} ，在给定观察变量条件下，计算状态变量的后验概率 $p(X|O, \theta^{\text{old}})$ ，然后计算对数 complete-data^[1] 似然在状态变量后验概率下的期望，定义为：

$$Q(\theta, \theta^{\text{old}}) = E_{\theta^{\text{old}}} [\ln p(X, O | \theta)] = \sum_X p(X | O, \theta^{\text{old}}) \ln p(X, O | \theta) \quad (14.10)$$

利用期望 $E[aX + bY + c] = aE[X] + bE[Y] + c$ 性质，可以得到：

$$\begin{aligned} E_{\theta^{\text{old}}} [\ln p(X, O | \theta)] &= E_{\theta^{\text{old}}} [\ln \{ p(x_1 | \pi) [\prod_{t=2}^T p(x_t | x_{t-1}, \mathbf{A})] \prod_{t=1}^T p(o_t | x_t) \}] \quad (14.11) \\ &= E_{\theta^{\text{old}}} \left[\sum_{j=1}^K x_{1j} \ln \pi_j + \sum_{t=2}^T \sum_{i=1}^K \sum_{j=1}^K x_{t-1,i} x_{tj} \ln a_{ij} + \right. \\ &\quad \left. \sum_{t=1}^T \sum_{j=1}^K x_{tj} \ln p(o_t | \phi_j) \right] \\ &= \sum_{j=1}^K E_{\theta^{\text{old}}} [x_{1j}] \ln \pi_j + \sum_{t=2}^T \sum_{i=1}^K \sum_{j=1}^K E_{\theta^{\text{old}}} [x_{t-1,i} x_{tj}] \ln a_{ij} + \\ &\quad \sum_{t=1}^T \sum_{j=1}^K E_{\theta^{\text{old}}} [x_{tj}] \ln p(o_t | \phi_j) \end{aligned}$$

令 $\gamma(x_{tj}) = E_{\pi_j^{\text{old}}} [x_{tj}]$ ， $\xi(x_{t-1,i}, x_{tj}) = E_{a_{ij}^{\text{old}}} [x_{t-1,i} x_{tj}]$ ，由于 $x_{tj} \in \{0, 1\}$ ，所以

$$\gamma(x_{tj}) = p(x_{tj} = 1 | \theta^{\text{old}}, O) \quad (14.12)$$

同样, 由于 $x_{t-1,i}x_{tj} \in \{0, 1\}$, 所以

$$\xi(x_{t-1,i}, x_{tj}) = p(x_{t-1,i} = 1, x_{tj} = 1 | \theta^{\text{old}}, O) \quad (14.13)$$

代入式 (14.11), 得到:

$$\begin{aligned} Q(\theta, \theta^{\text{old}}) &= E_{\theta^{\text{old}}}[\ln p(X, Q | \theta)] \\ &= \sum_{j=1}^K \gamma(x_{1j}) \ln \pi_j + \sum_{t=2}^T \sum_{i=1}^K \sum_{j=1}^K \xi(x_{t-1,i}, x_{tj}) \ln a_{ij} + \\ &\quad \sum_{t=1}^T \sum_{j=1}^K \gamma(x_{tj}) \ln p(o_t | \phi_j) \end{aligned} \quad (14.14)$$

后面我们将介绍如何高效计算上面的后验概率, 现在先假定它们是已知量。

M-Step: 对期望值求最大值 $Q(\theta, \theta^{\text{old}})$, 得到新的参数 $\theta^{\text{new}} = \arg \max_{\theta} Q(\theta, \theta^{\text{old}})$ 。

式 (14.14) 中与三个参数相关的部分是独立的, 所以参数 π, A, ϕ 可以分开求解:

$$\hat{\pi} = \arg \max_{\pi} \sum_{j=1}^K \gamma(x_{1j}) \ln \pi_j \quad (14.15)$$

$$\hat{A} = \arg \max_A \sum_{t=2}^T \sum_{i=1}^K \sum_{j=1}^K \xi(x_{t-1,i}, x_{tj}) \ln a_{ij} \quad (14.16)$$

$$\hat{\Phi} = \arg \max_{\Phi} \sum_{t=1}^T \sum_{j=1}^K \gamma(x_{tj}) \ln p(o_t | \phi_j) \quad (14.17)$$

式 (14.15) 就是 K 软分类问题 (交叉熵), 分类标签为 $\gamma(x_{1j})$, 预测值为 π_j 。但是式 (14.15) 优化还要加上约束条件 $\sum_j \pi_j = 1$, 可以使用拉格朗日乘子, 把约束问题改成如下增广目标函数^[2]:

$$L_{\pi} = \sum_{j=1}^K \gamma(x_{1j}) \ln \pi_j + \lambda(1 - \sum_j \pi_j) \quad (14.18)$$

直接对 π_k 求导等于 0 (KKT 条件), 得到:

$$\hat{\pi}_j = \frac{\gamma(x_{1j})}{\sum_{i=1}^K \gamma(x_{1i})} \quad (14.19)$$

同理，式 (14.16) 是 K^2 软分类问题（交叉熵），分类标签为 $\xi(x_{t-1,i}, x_{tj})$ ，预测值为 a_{ij} 。同样可以使用拉格朗日乘子，对 a_{ij} 求导等于 0，得到：

$$\hat{a}_{ij} = \frac{\sum_{t=2}^T \xi(x_{t-1,i}, x_{tj})}{\sum_{t=2}^T \sum_{l=1}^K \xi(x_{t-1,i}, x_{tl})} \quad (14.20)$$

式 (14.17) 就是样本带权对数似然，其实也就是普通混合模型 M-Step 中要求解的函数形式，HMM 就是多了一个转移概率式 (14.15) 求解和初始概率式 (14.16) 求解的步骤。不同的问题可以使用不同的发射概率，如果是高斯分布的话， $\phi_j = \{\mu_j, \Sigma_j\}$ ，那么可以得到如下闭式解：

$$\hat{\mu}_j = \frac{\sum_{t=1}^T \gamma(x_{tj}) o_t}{\sum_{t=1}^T \gamma(x_{tj})} \quad (14.21)$$

$$\hat{\Sigma}_j = \frac{\sum_{t=1}^T \gamma(x_{tj}) (o_t - \hat{\mu}_j)(o_t - \hat{\mu}_j)^T}{\sum_{t=1}^T \gamma(x_{tj})} \quad (14.22)$$

可以看出， $\hat{\mu}_j$ 其实就是观察变量带权均值， $\hat{\Sigma}_j$ 是带权样本方差。

综上所述，混合模型的 EM 求解过程，其实就是对样本属于不同混合组件权重重新计算、迭代的过程，即每个样本分成 K 个分数样本，组件 j 的优化只使用属于自己的样本，如图 14-6 所示。

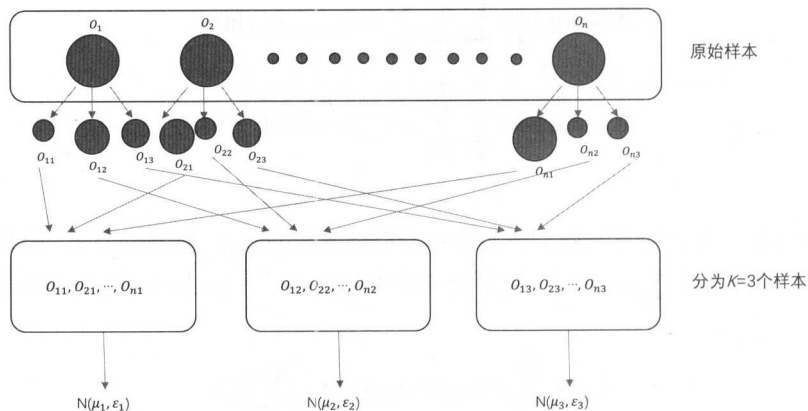


图 14-6 混合高斯 EM 算法样本重新划分，图中圆圈的大小代表样本权重大小

14.2.5 后验概率

上面我们介绍了在 HMM 中如何使用 EM 算法来对参数求解, 在求解的过程中假设后验概率 $\gamma(x_{tk}), \xi(x_{t-1,i}, x_{tj})$ 是已知的, 如果是简单的非时序混合模型, 那么由于不同的观察数据是独立同分布的, 后验概率只使用当前样本, 可以直接利用贝叶斯公式求解:

$$p(x_{nk} = 1 | o_n) = \frac{p(o_n | x_{nk} = 1)p(x_{nk} = 1)}{\sum_j p(o_n | x_{nj} = 1)p(x_{nj} = 1)} \quad (14.23)$$

HMM 中一条时序数据在不同时刻的观察数据不满足独立同分布的性质 (注意区别条件独立), 即任意时刻的观察数据都与之前时刻的观察数据相关, 因此任意时刻的状态后验概率 $\gamma(x_{tk})$ 不仅与当前时刻 t 的观察数据 o_t 相关, 而且还与整条时序数据在任意时刻的观察变量相关。同样, 转移后验概率 $\xi(x_{t-1,i}, x_{tj})$ 不仅与 $t-1, t$ 相邻时刻的观察数据 o_{t-1}, o_t 相关, 而且还与整条时序数据在任意时刻的观察变量相关。可以想象, HMM 后验概率的求解变得非常复杂。

为了表示方便, 以后使用 x_{tj} 代表 $x_{tj} = 1$, 表示 t 时刻的状态是 j , 即 $p(x_{tj} | x_{t-1,i})$ 表示 $p(x_{tj} = 1 | x_{t-1,i} = 1)$, $p(o_t | x_{tj})$ 表示 $p(o_t | x_{tj} = 1)$ 。使用 $b_j(o_t)$ 表示 t 时刻状态为 j 的发射概率, 即 $b_j(o_t)$ 表示 $p(o_t | x_{tj} = 1)$ 。

14.2.6 前向-后向算法

虽然求解后验概率比较复杂, 但是 HMM 概率图模型要比没有独立性假设的概率图模型简单很多, 最后能得到线性时间复杂度的算法。

我们先推导状态后验概率 $\gamma(x_{tj})$ 的求解过程。利用贝叶斯公式展开:

$$\gamma(x_{tj}) = p(x_{tj} | \theta^{\text{old}}, O) = \frac{p(O, x_{tj})}{p(O)} \quad (14.24)$$

如果直接对式 (14.24) 暴力展开求解后验概率的话, 复杂度将呈指数级增加, 因此这种方法不可取。

利用条件独立性质及 d-separation 准则, 得到:

$$\gamma(x_{tj}) = \frac{p(O | x_{tj})p(x_{tj})}{P(O)} \quad (14.25)$$

$$\begin{aligned}
&= \frac{p(o_{1:t}|x_{tj})p(o_{t+1:T}|x_{tj})p(x_{tj})}{p(O)} \\
&= \frac{p(o_{1:t}, x_{tj})p(o_{t+1:T}|x_{tj})}{p(O)} \\
&= \frac{\alpha(t, j)\beta(t, j)}{\sum_{i=1}^K \alpha(t, i)\beta(t, i)}
\end{aligned}$$

其中, $\alpha(t, j) = p(o_{1:t}, x_{tj})$, 表示观察变量 $o_{1:t}$ 和 t 时刻状态为 j 的联合概率分布; $\beta(t, j) = p(o_{t+1:T}|x_{tj})$, 表示 t 时刻给定状态为 j 的观察变量 $o_{t+1:T}$ 的条件概率分布。

根据条件独立性质, $\alpha(t, j), \beta(t, j)$ 满足动态规划法的最优子结构, 因此能得到如下动态规划递归求解:

$$\alpha(t, j) = b_j(o_t) \sum_{i=1}^K \alpha(t-1, i) a_{ij} \quad (14.26)$$

$$\beta(t, j) = \sum_{i=1}^K a_{ji} b_i(o_{t+1}) \beta(t+1, i) \quad (14.27)$$

下面分别给出如何利用条件独立性质及 d-separation 准则证明上面的递推式。

$$\begin{aligned}
\alpha(t, j) &= p(o_{1:t}, x_{tj}) \quad (14.28) \\
&= p(o_{1:t}|x_{tj})p(x_{tj}) \\
&= p(o_{1:t-1}|x_{tj}, o_t)p(o_t|x_{tj})p(x_{tj}) \\
&= p(o_{1:t-1}|x_{tj})p(o_t|x_{tj})p(x_{tj}) \\
&= p(o_t|x_{tj})p(o_{1:t-1}, x_{tj}) \\
&= p(o_t|x_{tj}) \sum_{i=1}^K p(o_{1:t-1}, x_{t-1,i}, x_{tj}) \\
&= p(o_t|x_{tj}) \sum_{i=1}^K p(o_{1:t-1}, x_{tj}|x_{t-1,i})p(x_{t-1,i}) \\
&= p(o_t|x_{tj}) \sum_{i=1}^K p(o_{1:t-1}|x_{t-1,i})p(x_{tj}|x_{t-1,i})p(x_{t-1,i}) \\
&= p(o_t|x_{tj}) \sum_{i=1}^K p(o_{1:t-1}, x_{t-1,i})p(x_{tj}|x_{t-1,i})
\end{aligned}$$

$$\begin{aligned}
&= b_j(o_t) \sum_{i=1}^K \alpha(t-1, i) a_{ij} \\
\beta(t, j) &= p(o_{t+1:T} | x_{tj}) \quad (14.29) \\
&= \sum_{i=1}^K p(o_{t+1:T}, x_{t+1,i} | x_{tj}) \\
&= \sum_{i=1}^K p(x_{t+1,i} | x_{tj}) p(o_{t+1:T} | x_{t+1,i}, x_{tj}) \\
&= \sum_{i=1}^K p(x_{t+1,i} | x_{tj}) p(o_{t+1:T} | x_{t+1,i}) \\
&= \sum_{i=1}^K p(x_{t+1,i} | x_{tj}) p(o_{t+1} | x_{t+1,i}) p(o_{t+2:T} | x_{t+1,i}, o_{t+1}) \\
&= \sum_{i=1}^K p(x_{t+1,i} | x_{tj}) p(o_{t+1} | x_{t+1,i}) p(o_{t+2:T} | x_{t+1,i}) \\
&= \sum_{i=1}^K a_{ji} b_i(o_{t+1}) \beta(t+1, i)
\end{aligned}$$

从图 14-7 可以看出, $\alpha(t, j)$ 是多条长度为 $t-1$ 的概率路径累加值, 它只与 $t-1$ 时刻的 $\alpha(t-1, i)$ 值有关, 与路径如何从前往后到达 $\alpha(t-1, i)$ 无关。同理, $\beta(t, j)$ 也是多条长度为 $T-t$ 的概率路径累加值, 它只与 $t+1$ 时刻的 $\beta(t+1, i)$ 值有关, 与路径如何从后往前到达 $\beta(t+1, i)$ 无关。前向概率递归的初始条件为 $\alpha(1, j) = \pi_j b_j(o_1)$, 后向概率递归的初始条件为 $\beta(T, j) = 1$ 。

前向-后向算法的计算复杂度为 $O(K^2T)$, 空间复杂度为 $O(KT)$, 时间和空间都是随时间呈线性增长的, 相比于指数级的暴力穷举算法有了很大提升。

同样, 利用贝叶斯公式和条件独立性质及 d-separation 准则来推导计算另一个后验概率 $\xi(x_{t-1,i}, x_{tj})$:

$$\begin{aligned}
\xi(x_{t-1,i}, x_{tj}) &= p(x_{t-1,i}, x_{tj} | O) \quad (14.30) \\
&= \frac{p(x_{t-1,i}, x_{tj}, O)}{p(O)} \\
&= \frac{p(o_{1:t-1}, o_t, o_{t+1:T} | x_{t-1,i}, x_{tj}) p(x_{t-1,i}, x_{tj})}{p(O)}
\end{aligned}$$

$$\begin{aligned}
&= \frac{p(o_{1:t-1}|x_{t-1,i})p(o_t|x_{tj})p(o_{t+1,T}|x_{tj})p(x_{tj}|x_{t-1,i})p(x_{t-1,i})}{p(O)} \\
&= \frac{p(o_{1:t-1}, x_{t-1,i})p(o_t|x_{tj})p(o_{t+1,T}|x_{tj})p(x_{tj}|x_{t-1,i})}{p(O)} \\
&= \frac{\alpha(t-1, i)b_j(o_t)a_{ij}\beta(t, j)}{p(O)}
\end{aligned}$$

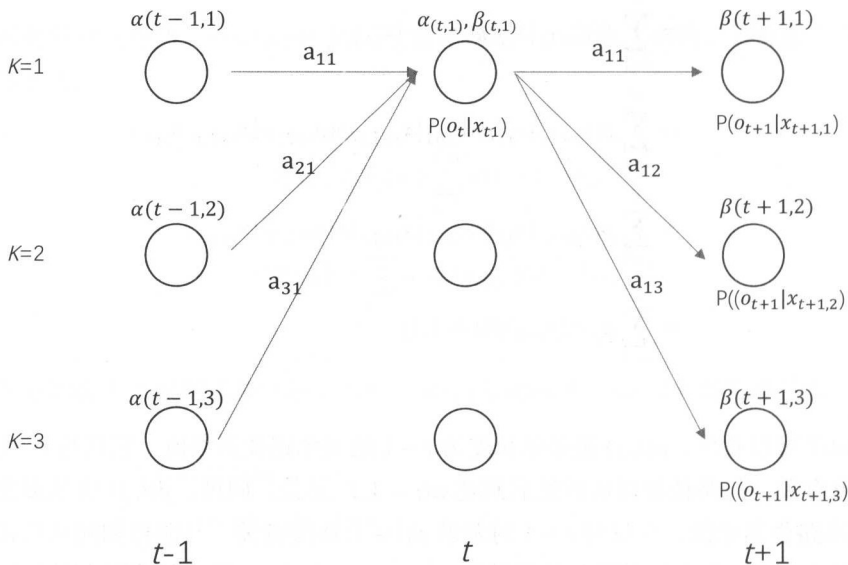


图 14-7 前向-后向算法

从上式可以看出，在计算前向-后向算法的同时，后验概率 $\xi(x_{t-1,i}, x_{tj})$ 的计算也可以顺便完成。最后把式 (14.25)、式 (14.30) 代入式 (14.19)、式 (14.20)、式 (14.21)、式 (14.22) 中就能得到完整的 EM 参数更新。

14.3 HMM 梯度求解

HMM 最常用的求解算法是基于 EM 迭代的，使用 EM 算法有如下原因。

- (1) 对数 log 里面是相加的形式，求导时没有对数里面是相乘的形式简单。

(2) 数据似然的概率路径数量跟时间呈指数级增长, $O(K^T)$, 直接展开枚举求导速度特别慢。

(3) HMM 是非凸函数, 对数 complete-data 似然在状态变量后验概率下的期望一般是凸函数, 而且一般能直接得到闭式解。

EM 算法虽然简单、有效, 但是在很多情况下没有数值迭代算法收敛快, 而且一旦梯度能快速求出, 就能使用拟牛顿等快速收敛算法。下面我们介绍两种 HMM 梯度快速求解的算法。

14.3.1 梯度算法 1

长度为 T 的序列数据的负对数似然根据乘法规则分解如下:

$$\begin{aligned} L &= -\ln p(O) = -\ln \prod_t^T p(o_t | o_{1:t-1}) \\ &= -\sum_t^T \ln p(o_t | o_{1:t-1}) \end{aligned} \quad (14.31)$$

引入隐变量, 利用条件独立性质及 d-separation 准则:

$$\begin{aligned} L &= -\sum_t^T \ln p(o_t | o_{1:t-1}) \\ &= -\sum_{t=1}^T \ln \sum_{j=1}^K p(o_t, x_{tj} | o_{1:t-1}) \\ &= -\sum_{t=1}^T \ln \sum_{j=1}^K p(o_t | x_{tj}, o_{1:t-1}) p(x_{tj} | o_{1:t-1}) \\ &= -\sum_{t=1}^T \ln \sum_{j=1}^K p(o_t | x_{tj}) p(x_{tj} | o_{1:t-1}) \\ &= -\sum_{t=1}^T \ln \sum_{j=1}^K b_j(o_t) \omega(t, j) \end{aligned} \quad (14.32)$$

其中, 后验概率 $p(x_{tj} | o_{1:t-1})$ 记为 $\omega(t, j)$, 求和 $\sum_i^K b_i(o_t) \omega(t, i)$ 记为 c_t , 对式 (14.32) 两

边求导，得到：

$$\frac{\partial L}{\partial \theta} = - \sum_{t=1}^T \frac{1}{c_t} \ln \left\{ \sum_{j=1}^K \frac{\partial b_j(o_t)}{\partial \theta} \omega(t, j) + b_j(o_t) \frac{\partial \omega(t, j)}{\partial \theta} \right\} \quad (14.33)$$

再利用条件独立性及 d-separation 准则，可以得到 $\omega(t, j)$ 的递推式：

$$\begin{aligned} \omega(t, j) &= p(x_{tj} | o_{1:t-1}) \\ &= \frac{p(o_{1:t-1}, x_{tj})}{\sum_{k=1}^K p(o_{1:t-1}, x_{tk})} \\ &= \frac{\sum_{i=1}^K p(o_{1:t-1}, x_{t-1,i}, x_{tj})}{\sum_{i=1}^K \sum_{k=1}^K p(o_{1:t-1}, x_{t-1,i}, x_{tk})} \\ &= \frac{\sum_{i=1}^K p(x_{tj} | x_{t-1,i}) p(o_{1:t-1}, x_{t-1,i})}{\sum_{i=1}^K \sum_{k=1}^K p(x_{tk} | x_{t-1,i}) p(o_{1:t-1}, x_{t-1,i})} \\ &= \frac{\sum_{i=1}^K p(x_{tj} | x_{t-1,i}) p(o_{1:t-1}, x_{t-1,i})}{\sum_{i=1}^K p(o_{1:t-1}, x_{t-1,i}) \underbrace{\sum_{k=1}^K p(x_{tk} | x_{t-1,i})}_{=1}} \\ &= \frac{\sum_{i=1}^K p(x_{tj} | x_{t-1,i}) p(o_{1:t-1}, x_{t-1,i})}{\sum_{i=1}^K p(o_{1:t-1}, x_{t-1,i})} \\ &= \frac{\sum_{i=1}^K p(x_{tj} | x_{t-1,i}) p(o_{t-1} | x_{t-1,i}) p(o_{1:t-2}, x_{t-1,i})}{\sum_{i=1}^K p(o_{t-1} | x_{t-1,i}) p(o_{1:t-2}, x_{t-1,i})} \\ &= \frac{\sum_i a_{ij} b_i(o_{t-1}) [p(o_{1:t-2}, x_{t-1,i}) / \sum_{j=1}^K p(o_{1:t-2}, x_{t-1,j})]}{\sum_i b_i(o_{t-1}) [p(o_{1:t-2}, x_{t-1,i}) / \sum_{j=1}^K p(o_{1:t-2}, x_{t-1,j})]} \\ &= \frac{\sum_i a_{ij} b_i(o_{t-1}) \omega(t-1, i)}{\sum_i b_i(o_{t-1}) \omega(t-1, i)} \\ &= \frac{1}{c_{t-1}} \sum_{i=1}^K a_{ij} b_i(o_{t-1}) \omega(t-1, i) \end{aligned} \quad (14.34)$$

对 $\omega(t, j)$ 递推式 (14.34) 两边同时求导，得到求导的递推式：

$$\begin{aligned} \frac{\partial \omega(t, j)}{\partial \theta} &= \frac{1}{c_{t-1}^2} \left\{ \left\{ \sum_{i=1}^K \frac{\partial a_{ij}}{\partial \theta} b_i(o_{t-1}) \omega(t-1, i) + a_{ij} \frac{\partial b_i(o_{t-1})}{\partial \theta} \omega(t-1, i) + a_{ij} b_i(o_{t-1}) \frac{\partial \omega(t-1, i)}{\partial \theta} \right\} \cdot c_{t-1} - \right. \\ &\quad \left. \left\{ \sum_{i=1}^K \frac{\partial b_i(o_{t-1})}{\partial \theta} \omega(t-1, i) + b_i(o_{t-1}) \frac{\partial \omega(t-1, i)}{\partial \theta} \right\} \cdot \left\{ \sum_{i=1}^K a_{ij} b_i(o_{t-1}) \omega(t-1, i) \right\} \right\} \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{c_{t-1}} \left\{ \left\{ \sum_{i=1}^K \frac{\partial a_{ij}}{\partial \theta} b_i(o_{t-1}) \omega(t-1, i) + a_{ij} \frac{\partial b_i(o_{t-1})}{\partial \theta} \omega(t-1, i) + a_{ij} b_i(o_{t-1}) \frac{\partial \omega(t-1, i)}{\partial \theta} \right\} - \right. \\
&\quad \left. \left\{ \sum_{i=1}^K \frac{\partial b_i(o_{t-1})}{\partial \theta} \omega(t-1, i) + b_i(o_{t-1}) \frac{\partial \omega(t-1, i)}{\partial \theta} \right\} \cdot \omega(t, j) \right\} \quad (14.35)
\end{aligned}$$

如果 θ 为发射概率参数, 那么递推式 (14.35) 可以简化如下:

$$\frac{\partial \omega(t, j)}{\partial \theta} = \frac{1}{c_{t-1}} \sum_{i=1}^K (a_{ij} - \omega(t, j)) \cdot \left\{ \frac{\partial b_i(o_{t-1})}{\partial \theta} \omega(t-1, i) + b_i(o_{t-1}) \frac{\partial \omega(t-1, i)}{\partial \theta} \right\} \quad (14.36)$$

如果 θ 为转移概率, 那么递推式 (14.35) 可以简化如下:

$$\frac{\partial \omega(t, j)}{\partial \theta} = \frac{1}{c_{t-1}} \sum_{i=1}^K b_i(o_{t-1}) \cdot \left\{ (a_{ij} - \omega(t, j)) \frac{\partial \omega(t-1, i)}{\partial \theta} + \omega(t-1, i) \frac{\partial a_{ij}}{\partial \theta} \right\} \quad (14.37)$$

使用梯度算法求解时, 也需要满足 $\sum_j a_{ij} = 1$ 的约束, 可以使用下面的变量替换把约束问题转变成非约束优化:

$$a_{ij} = \frac{e^{u_{ij}}}{\sum_{k=1}^K e^{u_{ik}}} \quad (14.38)$$

通过以上步骤可以看出, 求解后验概率 $\omega(t, j)$ 及其后验概率梯度 $\frac{\partial \omega(t, j)}{\partial \theta}$, 时间复杂度为 $O(K^2T)$, 空间复杂度为 $O(KT)$, 因此梯度算法 1 的复杂度跟前面介绍的前向-后向算法的复杂度一样。

14.3.2 梯度算法 2

在状态序列 X 展开, 负对数似然求导:

$$\begin{aligned}
\frac{\partial L}{\partial \theta} &= \frac{\partial -\ln p(O)}{\partial \theta} \\
&= \frac{\partial -\ln \sum_X p(X, O|\theta)}{\partial \theta} \\
&= - \frac{\sum_X \frac{\partial p(X, O|\theta)}{\partial \theta}}{p(X)} \quad (14.39)
\end{aligned}$$

从上式分子可以看出, 如果完全暴力展开求导, 则需要对 $O(K^T)$ 条概率路径求导 $\frac{\partial p(X, O|\theta)}{\partial \theta}$,

显然这种指数级的时间复杂度是不可行的。

但是如果针对具体某个变量求导，例如转移概率 $p(x_{tj}|x_{t-1,i})$ ，我们只需要对包含该变量的路径求导，但是包含转移概率 $p(x_{tj}|x_{t-1,i})$ 的路径数量还是指数级的，暴力求导还是不可行，但是我们发现进入 $x_{t-1,i}$ 之前的路径可以合并，合并的概率值就是 $\alpha(t-1, i)$ ，从 x_{tj} 出去后的路径也是可以合并的，合并的概率值就是 $\beta(t, j)$ ，如图 14-8 所示。

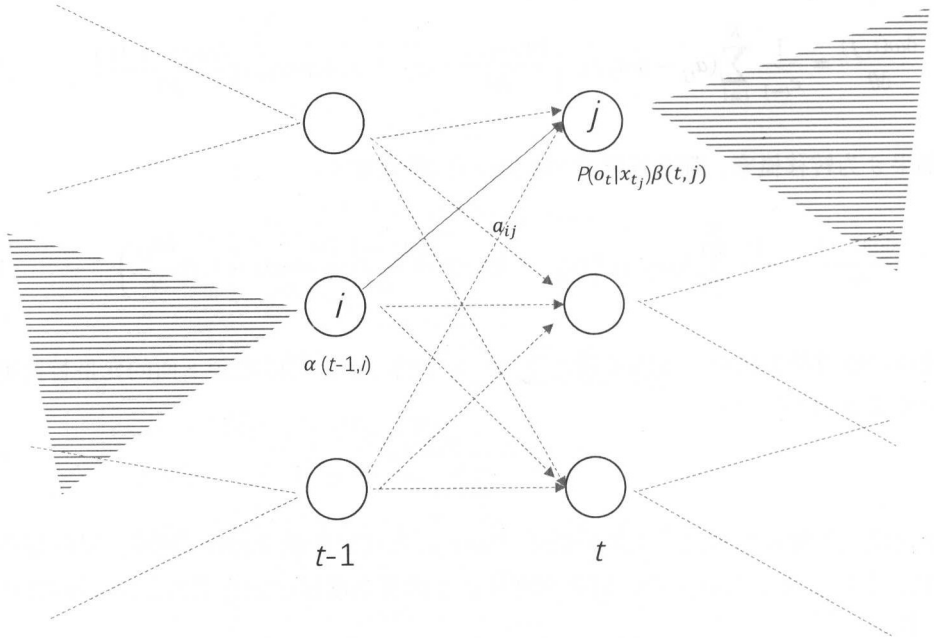


图 14-8 转移概率求导图

以上对转移概率 $p(x_{tj}|x_{t-1,i})$ 求导的描述过程可以用数学公式来展示：

$$\begin{aligned}
 \frac{\partial p(O)}{\partial p(x_{tj}|x_{t-1,i})} &= \frac{\partial \sum_{i'} \sum_{j'} p(x_{t-1,i'}, x_{tj'}, O)}{\partial p(x_{tj}|x_{t-1,i})} \\
 &= \frac{\partial p(x_{t-1,i}, x_{tj}, O)}{\partial p(x_{tj}|x_{t-1,i})} \\
 &= \frac{\partial p(o_{1:t-1}, x_{t-1,i}) p(o_t|x_{tj}) p(o_{t+1:T}|x_{tj}) p(x_{tj}|x_{t-1,i})}{\partial p(x_{tj}|x_{t-1,i})} \\
 &= p(o_{1:t-1}, x_{t-1,i}) p(o_t|x_{tj}) p(o_{t+1:T}|x_{tj}) \\
 &= \alpha(t-1, i) b_j(o_t) \beta(t, j)
 \end{aligned} \tag{14.40}$$

因为 $a_{ij} = p(x_{tj}|x_{t-1,i}), \forall t \in 2, \dots, T$, 所以

$$\begin{aligned} \frac{\partial p(O)}{\partial a_{ij}} &= \sum_{t=2}^T \frac{\partial p(O)}{\partial p(x_{tj}|x_{t-1,i})} \\ &= \sum_{t=2}^T \alpha(t-1, i) b_j(o_t) \beta(t, j) \end{aligned} \quad (14.41)$$

将式 (14.41) 代入式 (14.39) 中, 得到负对数似然对转移概率 a_{ij} 求导如下:

$$\begin{aligned} \frac{\partial L}{\partial a_{ij}} &= - \frac{\sum_{t=2}^T \alpha(t-1, i) b_j(o_t) \beta(t, j)}{p(O)} \\ &= \sum_{t=2}^T \frac{\alpha(t-1, i) b_j(o_t) \beta(t, j)}{p(O)} \\ &= \frac{\sum_{t=2}^T \xi(x_{t-1,i}, x_{tj})}{a_{ij}} \end{aligned} \quad (14.42)$$

细心的读者可能会有疑问, 如果一条概率路径上存在多个相同的转移概率 a_{ij} , 那么以上求导公式会重复或遗漏吗? 答案是不会! 我们可以使用如下两个方法证明。

(1) 如果把每个时刻的 $p(x_{tj}|x_{t-1,i})$ 当成 a_{ij} 的不同函数, $f_t(a_{ij}) = p(x_{tj}|x_{t-1,i}) = a_{ij}$, 很明显, $\frac{\partial f_t(a_{ij})}{\partial a_{ij}} = 1$, 根据链式法则即可证明:

$$\begin{aligned} \frac{\partial L}{\partial a_{ij}} &= \frac{\partial -\ln p(O)}{\partial a_{ij}} \\ &= \sum_{t=2}^T \frac{\partial -\ln p(O)}{\partial f_t(a_{ij})} \frac{\partial f_t(a_{ij})}{\partial a_{ij}} \\ &= \frac{\sum_{t=2}^T \xi(x_{t-1,i}, x_{tj})}{a_{ij}} \end{aligned} \quad (14.43)$$

(2) 假设一条概率路径上 a_{ij} 重复出现 m 次, 则这条路径可以表示为 $a_{ij}^m \times p_{\text{left}}$, 其中 p_{left} 表示概率路径上除了 a_{ij} 剩下的因子乘积, 对该条概率路径求导, 从求导公式可以看出, 一条概率路径上 a_{ij} 重复出现 m 次, 则相当于 m 次只出现一次 a_{ij} 的求导, 证明完毕。

$$\frac{\partial a_{ij}^m \times p_{\text{left}}}{\partial a_{ij}} = m \times \frac{a_{ij}^{m-1} \times p_{\text{left}}}{a_{ij}} \quad (14.44)$$

使用同样的技巧，可以求得对发射概率 $b_i(t)$ 的求导公式，求导如图 14-9 所示。

$$\begin{aligned}
 \frac{\partial L}{\partial b_i(t)} &= \frac{\partial -\ln p(O)}{\partial b_i(t)} \\
 &= -\frac{\alpha(t, i)\beta(t, i)/b_i(t)}{p(O)} \\
 &= -\frac{\gamma(x_{ii})}{b_i(t)}
 \end{aligned} \tag{14.45}$$

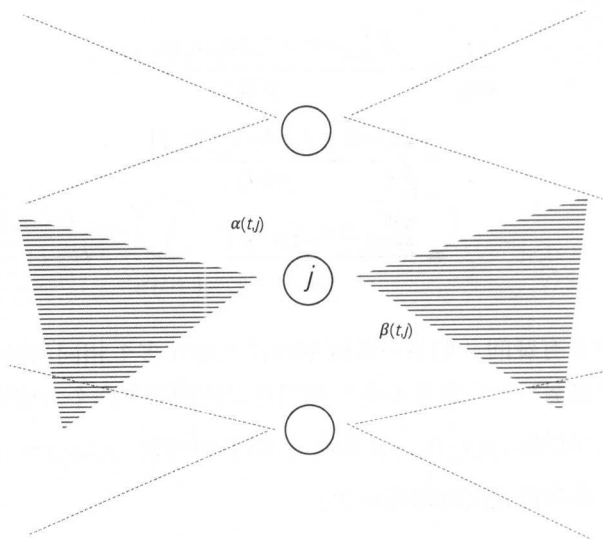


图 14-9 发射概率求导图

利用链式法则，可以得到对发射概率参数 ϕ_i 的求导公式：

$$\begin{aligned}
 \frac{\partial L}{\partial \phi_i} &= \frac{\partial -\ln p(O)}{\partial \phi_i} \\
 &= \sum_{t=1}^T \frac{\partial -\ln p(O)}{\partial b_i(t)} \frac{\partial b_i(t)}{\partial \phi_i} \\
 &= -\sum_{t=1}^T \frac{\gamma(x_{ii})}{b_i(t)} \frac{\partial b_i(t)}{\partial \phi_i}
 \end{aligned} \tag{14.46}$$

梯度算法 2 涉及了计算前向-后向算法中的 α, β ，因此时间、空间复杂度跟 EM 算法中的 E-Step 相同，快速、高效。

14.3.3 梯度求解的重要性

大部分资料中介绍 HMM 的参数求解方法时,只是提到 EM 算法,而我们这里介绍 HMM 梯度计算主要有如下原因。

(1) 梯度下降、拟牛顿等数值迭代优化算法一般都比 EM 算法快。

(2) 大部分资料中介绍的 HMM 都是静态的,在静态 HMM 中虽然数据是随着时间变化的,但是不同时间段的数据分布假设是相同的。而动态 HMM 数据分布是随时间变化的,这时候使用梯度算法更加容易。

(3) 在第 17 章中介绍的 CTC (Connectionist Temporal Classification),其参数求导公式基本跟梯度算法 2 一样,其实 CTC 本质上跟 HMM 一样,都是使用动态规划法来求解的,唯一的区别是 HMM 为似然 $p(O|W)$ 建模,而 CTC 直接为后验概率 $p(W|O)$ 建模,其中 W 为标注序列,所以读者要是理解了 HMM 的求导公式,那么理解 CTC 就很容易了。

14.4 孤立词识别

在介绍复杂的连续语音识别之前,我们先介绍一下孤立词识别 (Isolated Word Recognition)。在孤立词识别场景中,一条语音就只说一个词。孤立词识别虽然简单,但它是连续语音识别的基础,也有很多应用场景,比如电话拨号识别,主要识别 0,1,...,9 几个数字。这种专用的语音识别系统所需的训练数据更少、解码更快,最重要的是比通用的语音识别系统的识别精度更高。

14.4.1 特征提取

语音信号是时间维度上连续变化的一维模拟信号,在使用计算机处理之前,需要进行采样将其变成离散的数字信号,如图 14-10 所示。语音信号特征分析方法大致分为时域、频率两大类。时域信号分析方法具有简单易懂、物理意义明确、计算量小等特点。但是语音感知的很多特性都反映在功率谱中,语音信号的频谱相比于时域信号具有更加明显的声学特性,而且频率分析对于环境变化,鲁棒性更高,因此,频域分析方法在语音识别中使用更为广泛。

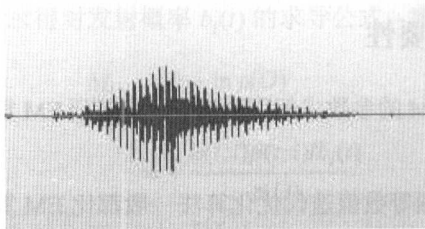


图 14-10 单词“one”的语音波形图

语音信号是随时间变化的，是一种非稳态的过程，但是口腔肌肉的运动频率相比于语音频率来说是缓慢的，在短时间内可以看作是一个稳态的过程。因此，为了同时保留语音信号时域和频域的信息，可以将语音信号分成一段一段来做短时加窗分析。窗口大小要正好合适，窗口越大，频域信息越精确，时域信息越不精确；反过来，窗口越小，时域信息越精确，频域信息越不精确。现在大多数语音识别系统中都使用 10ms 间隔（称为一帧），窗口大小为 25ms。

目前在语音识别系统中常用的语音特征有基于声道的特征模型：线性预测编码（LPC）、线性预测倒谱系数（LPCC）等，还有基于人耳听觉机理和人耳对声音频率感知的特征模型：梅尔频率倒谱系数（MFCC）。想了解更多的特征提取方法和处理细节，请参考文献 [11]。但是随着机器性能的提升，特别是 GPU 的出现，现在一些端到端的语音识别系统直接使用频谱信息来做深度卷积，省去了复杂的传统信号处理过程，而且往往能得到更好的识别精度。

14.4.2 孤立词建模

给定一段语音 $O = o_1, o_2, \dots, o_T$ ，孤立词识别的过程可以表示为：

$$\arg \max_i \{p(w_i|O)\} \quad (14.47)$$

其中 w_i 是字典里面的第 i 个词，利用贝叶斯公式转换成等价的问题：

$$\begin{aligned} \arg \max_i \{p(w_i|O)\} &= \arg \max_i \left\{ \frac{p(O|w_i)p(w_i)}{p(O)} \right\} \\ &= \arg \max_i \{p(O|w_i)p(w_i)\} \end{aligned} \quad (14.48)$$

其中似然 $p(O|w_i)$ 称为声学模型（Acoustic Model），先验概率 $p(w_i)$ 称为语言模型（Language Model）。一般语言模型可以直接从文本语料中统计得到，做孤立词识别时，假设所有

的词 w_i 出现的频率是相同的, 所以问题就变成了:

$$\arg \max_i \{p(w_i|O)\} = \arg \max_i \{p(O|w_i)\} \quad (14.49)$$

直接对似然概率 $p(O|w_i) = \prod_{t=1}^T p(o_t|o_{1:t-1}, w_i)$ 建模比较困难, 因为对于同一个词 w_i , 在不同的上下文环境中产生的语音长度不一样, 因此使用前面介绍的“弹簧模型”HMM 来为 $p(O|w_i)$ 建模非常合适。

使用 HMM 进行孤立词语音识别有两种可选方案, 其中第一种方案是字典里面的所有词都使用同一个 HMM; 第二种方案是每个词都使用单独的 HMM, 分开训练。

在第一种方案中, 所有的词都使用同一个 HMM 来训练、解码, 最大的优势就是简单, 但是可能需要的状态数量比较多, 因为不同的数字状态序列可能会大不相同。这样会造成状态转移矩阵非常大, 可能大部分状态之间的转移概率都几乎为零, 要照顾到任意两个状态直接的转移情况, 最终需要的训练数据就会特别多。而且由于 EM 算法和计算梯度的时间复杂度与状态数量的平方成正比, 所以这种方案是不可行的。

在第二种方案中, 针对每个词都训练一个 HMM, 即 $p(O|w_i) = p(O|\text{HMM}_i)$, 这样不同的词 HMM 之间的状态不会相互干扰。这种方案适合于字典特别小的场景, 例如上面提到的电话拨号识别。

在语音识别系统中使用的 HMM 跟前面介绍的通用的 HMM 稍有不同, 在通用的 HMM 模型中任意两个状态之间都有转移, 而在语音识别系统中转移概率是带约束的 left-right HMM: 状态只能转移到自己或者下一个固定的状态, 这是由发音是顺序的特点决定的 (当然, 也可以在不相邻的状态之间加一些跳边, 支持漏发的音素建模), 如图 14-11 所示。

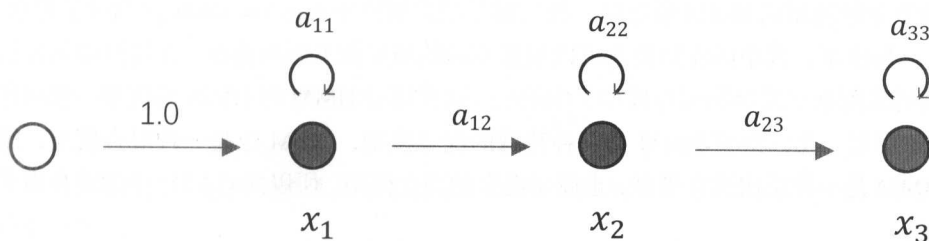


图 14-11 left-right HMM

孤立词识别的训练和解码流程非常简单, 如图 14-12 所示。首先为每个词收集足够的训练数据, 然后每个词单独训练自己的 HMM。识别就更加简单了, 使用前面介绍的前向算

法就能快速地计算所有孤立词 HMM 似然 $p(O|HMM_i) = \sum_{i=1}^K \alpha(T, i)$ ，然后选择似然最大的 HMM 模型对应的词作为识别结果。如果字典里面的词的数目为 m ，那么一次孤立词识别的时间复杂度为 $O(mK^2T)$ 。

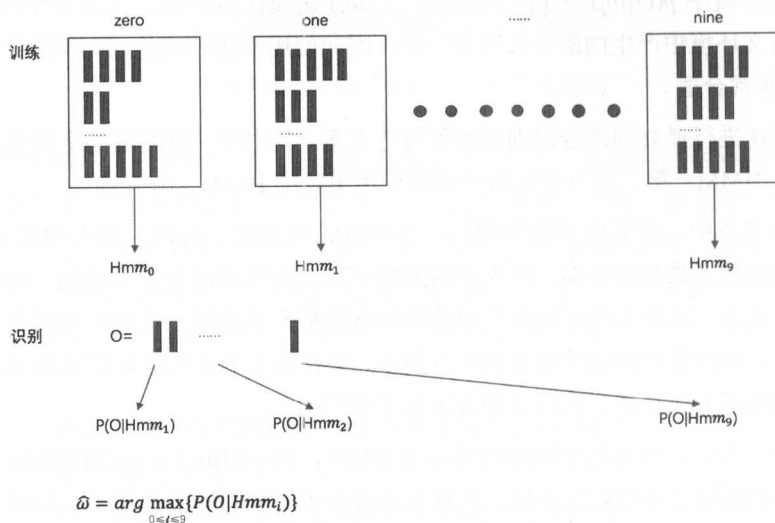


图 14-12 孤立词识别的训练和解码流程

14.4.3 GMM-HMM

前面章节介绍通用的 HMM 模型时，我们都是把发射概率 $b_j(o_t) = p(o_t|x_{tj})$ 当成了黑盒。HMM 只是一种框架，其中的发射概率可以根据不同的场景做不同的选择。在传统的语音识别中最常用的发射概率是高斯混合模型（GMM），这种结合 HMM 和 GMM 的模型一般叫作 GMM-HMM 模型。前面讲到 HMM 是一种特殊的混合模型，GMM 也是一种混合模型，因此 GMM-HMM 是一种层次混合模型，也就是混合的混合模型，所以式 (14.51) 中的混合因子 c_{jm} 是二维的。

$$b_j(o_t) = \sum_{z_j} p(o_t|z_j)p(z_j) \quad (14.50)$$

其中， z_j 是 HMM 第 j 个状态的隐变量，是 M 维的 one-hot 编码，即 $z_{jm} \in \{0, 1\}$ ，且

$\sum_{m=1}^M z_{jm} = 1$ 。z 先验概率使用多项式分布来建模:

$$p(z_j) = \prod_{m=1}^M c_{jm}^{z_{jm}} \quad (14.51)$$

其中, M 为 GMM 混合因子的数量, c_{jm} 为 GMM-HMM 第 j 个状态的第 m 个混合因子, 必须满足约束 $\sum_{m=1}^M c_{jm} = 1$ 。

类似地, 给定 z_j , o_t 的条件概率分布 $p(o_t|z_j)$ 可以表示为:

$$p(o_t|z_j) = \prod_{m=1}^M \mathcal{N}(o_t|\mu_{jm}, \Sigma_{jm})^{z_{jm}} \quad (14.52)$$

o_t, z_j 的联合概率分布可以表示为:

$$p(o_t, z_j) = \prod_{m=1}^M (c_{jm} \mathcal{N}(o_t|\mu_{jm}, \Sigma_{jm}))^{z_{jm}} \quad (14.53)$$

t 时刻观察变量的边际概率的分布 $p(o_t)$ 为:

$$\begin{aligned} p(o_t) &= \sum_{z_j} p(o_t, z_j) \\ &= \sum_{m=1}^M c_{jm} \mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm}) \end{aligned} \quad (14.54)$$

其中, $\mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm})$ 为多变量高斯分布:

$$\mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm}) = \frac{1}{\sqrt{(2\pi)^n |\Sigma_{jm}|}} e^{-\frac{1}{2}(o_t - \mu_{jm})^T \Sigma_{jm}^{-1} (o_t - \mu_{jm})} \quad (14.55)$$

其中, μ_{jm}, Σ_{jm} 为 HMM 第 j 个状态的第 m 个高斯的均值和协方差矩阵, n 为观察变量特征维数。

GMM-HMM 做 EM 参数估计时, 转移概率与发射概率的具体形式无关, 所以保持不变。相比于单高斯参数更新公式 (14.21) 和式 (14.22), GMM 的参数更新公式稍微有点不同:

$$\hat{\mu}_{jm} = \frac{\sum_{t=1}^T \gamma(x_{tjm}) o_t}{\sum_{t=1}^T \gamma(x_{tjm})} \quad (14.56)$$

$$\hat{\Sigma}_{jm} = \frac{\sum_{t=1}^T \gamma(x_{tjm})(o_t - \hat{\mu}_{jm})(o_t - \hat{\mu}_{jm})^T}{\sum_{t=1}^T \gamma(x_{tjm})} \quad (14.57)$$

其中, $\gamma(x_{tjm}) = E_{\theta^{\text{old}}}[x_{tjm}]$ 为 t 时刻 HMM 状态为 j 、GMM 状态为 m 的后验概率:

$$\begin{aligned} \gamma(x_{tjm}) &= E_{\theta^{\text{old}}}[x_{tjm}] \\ &= p(x_{tjm} = 1 | O, \theta^{\text{old}}) \\ &= p(z_{jm} = 1 | x_{tj} = 1, O, \theta^{\text{old}}) p(x_{tj} = 1 | O, \theta^{\text{old}}) \\ &= p(z_{jm} = 1 | x_{tj} = 1, O, \theta^{\text{old}}) \gamma(x_{tj}) \end{aligned} \quad (14.58)$$

其中, $\gamma(x_{tj})$ 是通过前向-后向算法式 (14.25) 得到的。在给定 $x_{tj} = 1$ 的情况下, z_{jm} 与观察变量 $o_{1:t-1}, o_{t+1:T}$ 的路径被阻塞, 即满足条件独立性质:

$$p(z_{jm} = 1 | x_{tj} = 1, O, \theta^{\text{old}}) = p(z_{jm} = 1 | x_{tj} = 1, o_t, \theta^{\text{old}}) \quad (14.59)$$

那么

$$\begin{aligned} \gamma(x_{tjm}) &= p(z_{jm} = 1 | x_{tj} = 1, o_t, \theta^{\text{old}}) \gamma(x_{tj}) \\ &= \frac{p(z_{jm}, o_t)}{\sum_{i=1}^M p(z_{im}, o_t)} \gamma(x_{tj}) \\ &= \frac{c_{jm} \mathcal{N}(o_t; \mu_{jm}, \Sigma_{jm})}{\sum_{s=1}^M c_{js} \mathcal{N}(o_t; \mu_{js}, \Sigma_{js})} \gamma(x_{tj}) \end{aligned} \quad (14.60)$$

GMM 中混合因子的更新公式类似于式 (14.15):

$$\hat{c}_{jm} = \frac{\sum_{t=1}^T \gamma(x_{tjm})}{\sum_{t=1}^T \sum_{s=1}^M \gamma(x_{tjs})} \quad (14.61)$$

GMM-HMM 层次混合模型包含了大量的参数, 具体的细节见表 14-1。

表 14-1 GMM-HMM 参数规模

参数	规模
HMM 转移概率	$O(K^2)$
初始状态概率	$O(K)$

续表

参数	规模
GMM 中混合因子	$O(KM)$
GMM 中均值	$O(KMn)$
GMM 中协方差矩阵	$O(KMn^2)$ 或者 $O(KMn)$ (对角阵)

14.5 连续语音识别

前面章节介绍的孤立词识别是语音识别中最简单的场景,在该场景中每条语音只包含一个词。本节我们将介绍更加复杂、更为通用的语音识别场景——连续语音识别,它能够对一段连续语音进行识别,语音中可能包含多个词,甚至静音、有背景噪音。

在孤立词识别场景中,为每个词都训练一个 HMM,借助同样的思想,在连续语音识别场景中,如果为每一段文本都建立一个 HMM,那么参数规模随句子长度呈指数级增长 $O(|W|^N)$,其中 $|W|$ 为词汇量, N 为句子长度,在词汇量稍大一点的场景中就会撑不住,最关键的是没有那么多语料来训练,即使把现在全人类的语音都用作训练数据也是杯水车薪。

我们可以利用搭积木的思想来解决语音识别模型的问题。例如,使用简单的几种乐高积木就可以自由灵活地拼接成各种各样的物体,如果对每种物体都定制专门的积木,那么制造成本就太高了。借助这种搭积木的思想,我们可以对字典中的每个词都建立一个 HMM,然后针对特定的文本序列 $W = w_1, w_2, \dots, w_L$, 句子级别的 HMM 就可以使用涉及单词的 HMM 拼接组装起来, $HMM_W = HMM_{w_1} HMM_{w_2} \dots HMM_{w_L}$ 。但是在大规模的连续语音识别场景中,字典往往包含大量的词汇,像英语词汇量有几十万量级,如果针对每个单词都建立一个 HMM,那么参数规模也会非常大。所以,一般的连续语音识别系统都是基于子词(sub-word)级别的 HMM 的,子词就是比词更加细粒度的单元,英语的子词可以是音素(phone)、字母的 n -gram,甚至可以是字母的瓦片(shingle);汉语的子词一般是声韵母。

在基于 HMM 的英语语音识别系统中,最常用的子词是音素,英语有 40 个左右的音素。如图 14-13 所示,由音素粒度 HMM 构建词粒度 HMM,再由词粒度 HMM 构建整个句子级别的 HMM,自底向上搭建了一个层次 HMM,整个语音识别系统就像搭积木一样构建起来。

参数规模问题解决了,那么现在该如何训练呢?在孤立词识别系统中,每条语音就涉及一个词,不需要做切分,不同词的 HMM 是分开训练的;而在连续语音识别系统中,训练数据每段语音可能涉及多个词,单词或子词之间的边界事先并不知道,人工切分的方案是不可行的,因为成本太高,速度太慢,而且受个人知识水平、当时的心情状态等因素影响,有时

候切得不一定准。当然，少量地切分数据也是有用的，在训练初始时做 bootstrap^[6]，可以加快收敛速度。

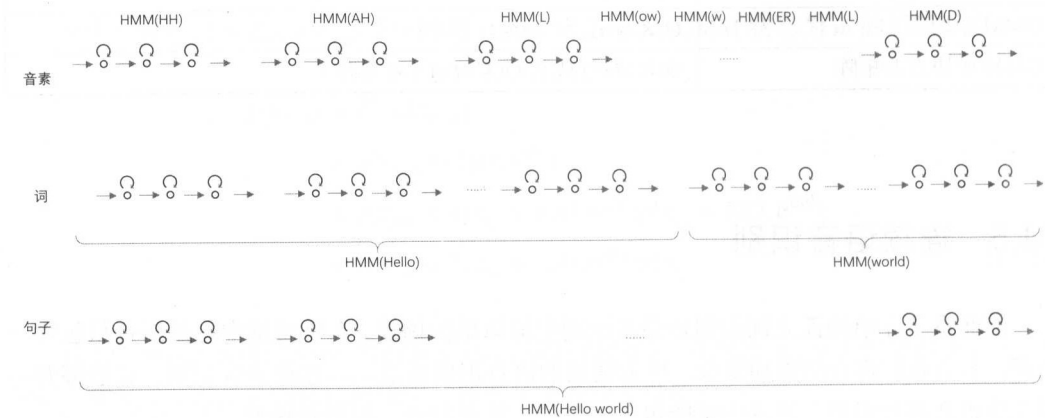


图 14-13 句子“Hello world”的层次 HMM

其实并不需要手工做硬切分,因为拼接的 HMM 也与普通的 HMM 一样,可以使用 Baum-Welch 算法对数据进行软切分 (E-Step 求后验概率的过程其实就是对数据进行软切分的过程),即任何时刻 t 都有可能出现任意单词或子词的任意状态 (当然开始状态和结束状态需要特殊对待),只不过出现的概率有大小之分,所以称之为软切分。这种训练方法也称为嵌入式训练 (Embedded Training)。同样,也可以使用 14.6 节介绍的 Viterbi 算法进行硬切分,那么后验概率 $\gamma(x_{ij}) \in \{0, 1\}$,该训练方法称为 Viterbi 训练。接下来我们将着重介绍嵌入式训练, Viterbi 训练的流程与之类似,不再赘述。

数据切分问题解决了,那么现在还剩一个问题:各个子词 HMM 是如何拼接的?为了解决 HMM 拼接问题,一般在子词 HMM 前后各加入一个非发射概率的状态,分别称为开始状态和结束状态,这两种状态不吸收语音帧,只起到衔接的作用,如图 14-14 所示,其中 x_0 为开始状态, x_4 为结束状态。

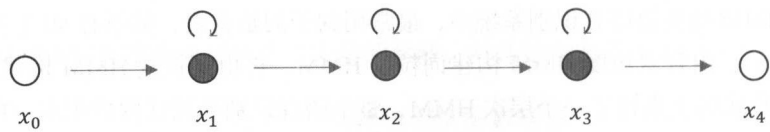


图 14-14 含开始状态的 HMM

开始状态转移到第一个状态的概率总是为 1.0,这是为了把前一个 HMM 的结束状态直接连接到当前 HMM 的第一个发射状态。但是结束状态的概率不为 1,它就是正常的转移概

率, 只不过该转移概率可以连接其他任意 HMM 的第一个状态。那么式 (14.26) 和式 (14.27) 所涉及的前向-后向计算公式只需要在开始状态和结束状态衔接处稍微改动一下:

$$\alpha^q(t, j) = \begin{cases} b_j^q(t) [\alpha^q(t-1, j-1) a_{j-1, j}^q + \alpha^q(t-1, j) a_{jj}^q], & \text{如果 } j \text{ 为第 } q \text{ 个 HMM 的发射状态,} \\ & \text{且不是第一个发射状态} \\ b_j^q(t) [\alpha^{q-1}(t-1, i) a_{if}^{q-1} + \alpha^q(t-1, j) a_{jj}^q], & \text{如果 } j \text{ 为第 } q \text{ 个 HMM 的第一个发射} \\ & \text{状态, 其中 } i \text{ 为第 } q-1 \text{ 个 HMM 的} \\ & \text{最后一个发射状态, } f \text{ 为第 } q-1 \\ & \text{个 HMM 的结束状态} \end{cases} \quad (14.62)$$

$$\beta^q(t, j) = \begin{cases} a_{ji}^q b_i^q(t+1) \beta^q(t+1, i) + a_{jj}^q b_j^q(t+1) \beta^q(t+1, j), & \text{如果 } j \text{ 为第 } q \text{ 个 HMM 的发射} \\ & \text{状态, 且不是最后一个发} \\ & \text{射状态} \\ a_f^q b_i^{q+1}(t+1) \beta^{q+1}(t+1, i) + a_{jj}^q b_j^q(t+1) \beta^q(t+1, j), & \text{如果 } j \text{ 为第 } q \text{ 个 HMM 的最后} \\ & \text{一个发射状态, } f \text{ 是第 } q \text{ 个} \\ & \text{HMM 的结束状态, } i \text{ 是第 } q+1 \\ & \text{个 HMM 的第一个发射状态} \end{cases} \quad (14.63)$$

前向概率的初始条件为 $\alpha^1(1, j) = b_j(o_1)$, 后向概率的初始条件为 $\beta^Q(T, j) = 1$ 。

同一个 HMM 之间的状态转移后验概率为:

$$\xi(x_{t-1, i}^q, x_{tj}^q) = \frac{\alpha^q(t-1, i) b_j^q(t) a_{ij}^q \beta^q(t, j)}{p(O)} \quad (14.64)$$

其中, i, j 都是第 q 个 HMM 的发射状态。

前后相邻 HMM 之间的状态转移后验概率为:

$$\xi(x_{t-1, i}^q, x_{tj}^{q+1}) = \frac{\alpha^q(t-1, i) b_j^{q+1}(t) a_f^q \beta^{q+1}(t, j)}{p(O)} \quad (14.65)$$

其中, i 是第 q 个 HMM 的最后一个发射状态, j 是第 $q+1$ 个 HMM 的第一个发射状态。

需要注意的是,在嵌入式训练求前向-后向时,每个子词的开始状态和结束状态是不参与递归计算的,它们只是起到一个逻辑上的衔接作用,只有结束转移概率 a_f^q 参与计算。

其中 EM 算法的参数估计跟之前的介绍基本相同,之前介绍的 HMM 参数估计只讨论了一条语音训练数据的情况,现在假设训练数据中总共有 R 条语音,那么基于子词级别的 GMM-HMM 嵌入式训练参数更新公式如下。

发射状态之间的转移概率更新公式:

$$\hat{a}_{ij}^q = \frac{\sum_{r=1}^R \sum_{t=1}^T \xi_r(x_{t-1,i}^q, x_{tj}^q)}{\sum_{r=1}^R \sum_{t=1}^T \sum_{l=1}^K \xi_r(x_{t-1,i}^q, x_{tl}^q)} \quad (14.66)$$

转移到结束状态的转移概率更新公式:

$$\hat{a}_f^q = \frac{\sum_{r=1}^R \sum_{t=1}^T \xi_r(x_{t-1,i}^q, x_{tj}^{q+1})}{\sum_{r=1}^R \sum_{t=1}^T \sum_{l=1}^K \xi_r(x_{t-1,i}^q, x_{tl}^{q+1})} \quad (14.67)$$

GMM 混合因子的更新公式:

$$\hat{c}_{jm} = \frac{\sum_{r=1}^R \sum_{t=1}^T \gamma_r(x_{tjm})}{\sum_{r=1}^R \sum_{t=1}^T \sum_{s=1}^M \gamma_r(x_{tjs})} \quad (14.68)$$

GMM 高斯分布均值更新公式:

$$\hat{\mu}_{jm} = \frac{\sum_{r=1}^R \sum_{t=1}^T \gamma_r(x_{tjm}) o_t^r}{\sum_{r=1}^R \sum_{t=1}^T \gamma_r(x_{tjm})} \quad (14.69)$$

GMM 高斯分布协方差更新公式:

$$\hat{\Sigma}_{jm} = \frac{\sum_{r=1}^R \sum_{t=1}^T \gamma_r(x_{tjm}) (o_t^r - \hat{\mu}_{jm})(o_t^r - \hat{\mu}_{jm})^T}{\sum_{r=1}^R \sum_{t=1}^T \gamma_r(x_{tjm})} \quad (14.70)$$

14.6 Viterbi 解码

在 14.2.4 节介绍的 Baum-Welch 算法中,计算观察数据似然时,考虑了所有可能的状态序列, $p(O) = \sum_X p(X, O)$, 因为从理论来讲,任意时刻都有可能出现任意状态。在所有的状态序

列中肯定有一个使得概率路径最大的状态序列 $X^* = \arg \max_X p(X, O) = \arg \max_X p(x_1) \prod_{t=2}^T p(x_t | x_{t-1}) \prod_{t=1}^T p(o_t | x_t)$ ，在大多数情况下，这个最大的概率路径的概率值占主宰，因此可以使用概率最大路径来做近似。

由于满足最优子结构，因此同样可以使用动态规划法来得到最优状态序列，只需要把前向-后向递归式 (14.26) 和式 (14.27) 中的求和 (sum) 换成求最大 (max) 即可：

$$\alpha_{\max}(t, j) = b_j(o_t) \max_i \alpha_{\max}(t-1, i) a_{ij} \quad (14.71)$$

初始条件：

$$\alpha_{\max}(1, j) = \pi_j b_j(o_1) \quad (14.72)$$

其中， $\alpha_{\max}(t, j)$ 表示观察到序列 $o_{1:t}$ 且 t 时刻的状态为 j 的最大似然。

那么，最终整个观察序列的最大似然概率值为：

$$\max_j \{\alpha(T, j) b_j(o_T)\} \quad (14.73)$$

把式 (14.71) 中每一时刻的最优状态记录下来，就能得到最优状态序列 X^* ，如图 14-15 所示，粗线代表的是最优状态序列路径。

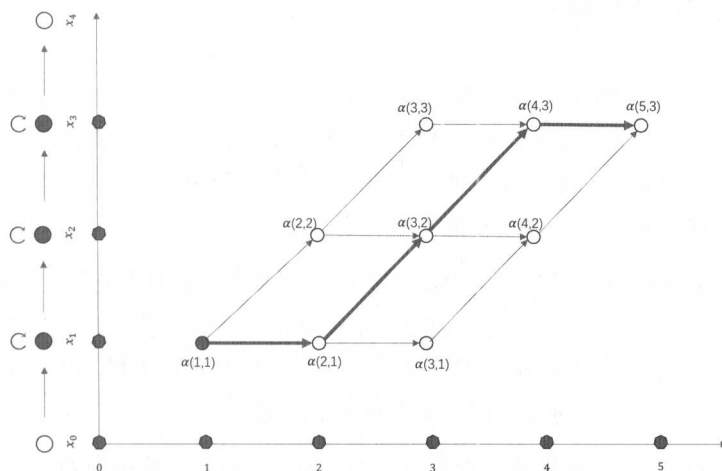


图 14-15 Viterbi 解码图，其中粗线表示最优路径

以上算法称之为 Viterbi 算法，我们经常把 Viterbi 算法解码的过程称为强制对齐。上面

只考虑单个 HMM 的 Viterbi 解码情况，同样，对于多个子词 HMM 连接的 Viterbi 解码，只需要把式 (14.62) 中的求和 (sum) 换成求最大 (max) 即可，这里就不赘述了。

我们使用 Baum-Welch 算法进行参数估计时考虑了所有状态序列的情况，而且计算时间复杂度跟 Viterbi 相同，那么 Viterbi 算法还有什么存在意义？下面我们简单介绍 Viterbi 算法的用途。

(1) 在训练时，除了使用 Baum-Welch 算法进行训练，也可以使用 Viterbi 算法进行参数估计，只不过要把后验概率 $\gamma(x_{tj}), \xi(x_{t-1,i}, x_{tj})$ 换成 0-1 指示函数： $\delta(x_{tj} = 1), \delta(x_{t-1,i} = 1, x_{tj} = 1)$ 。观察数据从软切分换成了硬切分。

(2) 最优状态序列得到以后，可以由状态映射到子词，再从子词映射到词，这样就可以得到子词之间的边界、词之间的边界。也就是说，知道了每个子词、词最可能的起始时间，我们就可以做很多事情，比如想知道用户某个音素的发音怎么样，就可以根据 Viterbi 算法解码得到该音素的边界，然后计算对应的似然就可以评估用户发音的好坏了。

(3) 在连续语音识别解码过程中，由于事先不知道文字序列，因此解码成任意文字序列都有可能，搜索状态空间呈指数级增长，需要结合 Viterbi 算法和 Beam Search 剪枝进行贪心近似解码，后面的章节中会详细介绍解码流程。

14.7 三音素状态聚类

前面介绍的子词级别的 HMM 都是基于单音素 (Monophone) 的，这种基于单音素的 HMM 是没有考虑上下文环境的，相同的音素，发音会因前后音素的影响而不同。如果不同上下文使用同一个高斯分布来建模，那么肯定精度不够，即使使用混合高斯分布来建模，在混合因子数量有限的情况下，非线性能力也还是有限的。

三音素 (Triphone) 模型的提出，是为了解决单音素没有考虑上下文影响的问题。单音素只考虑当前音素 q ，而三音素联合当前音素 q 和前后各一个音素 x, y ，一般标记为 $x-q+y$ ，三音素的思想跟 NLP 中的 3-gram (其实本质上都是三阶马尔可夫模型) 类似。三音素模型相比于单音素模型有很大的进步，但是会遇到如下问题。

(1) 参数规模过大：如果单音素数量为 Q ，那么三音素的数量为 Q^3 ，因此三音素的参数规模是单音素的 Q^2 倍。

(2) 过拟合：一般语音训练数据有限，这些数据涉及的三音素数量要远小于 Q^3 ，必然会造成训练数据拟合得很好，但是对于那些在训练数据中未见过的三音素效果会很差。

我们发现,对于每一个音素 q ,虽然有 Q^2 种不同的上下文,但是有很多上下文的发音规律是相同的,因此可以把一些发音规律相同的三音素做一次聚类,也叫绑定(Tying),把聚在同一个类下面的所有三音素当成同一种三音素,使用相同的参数。为了得到更好的效果,一般都在状态级别上做聚类(一般使用的都是3状态HMM),针对每一个音素的每种状态所涉及的所有上下文做聚类,最终聚在同一类的状态叫绑定状态(Tied-state),或叫共享状态,如图14-16所示。

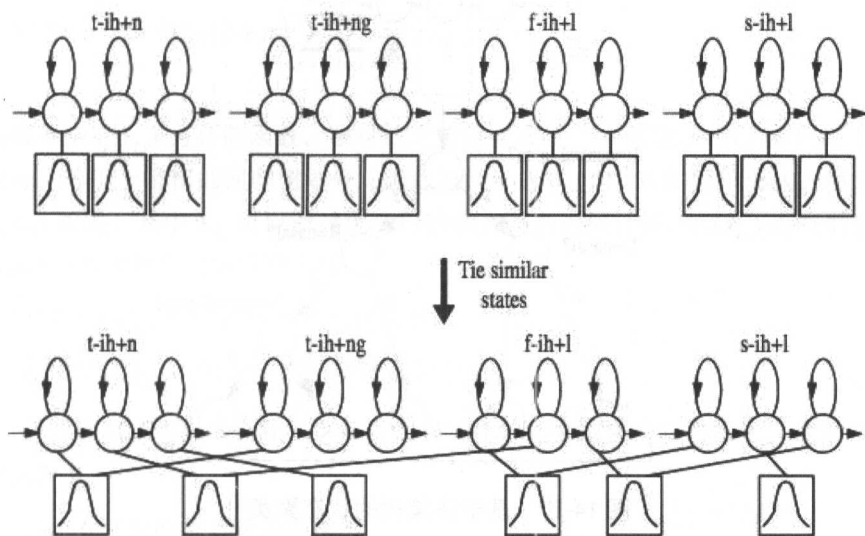
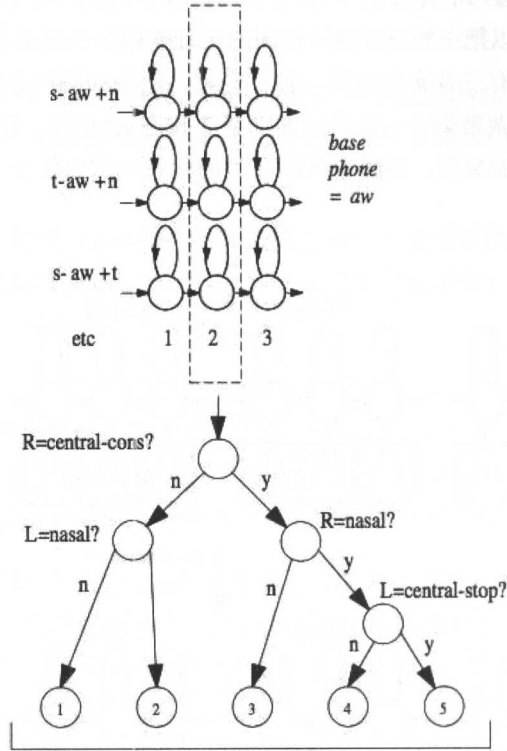


图 14-16 三音素的绑定状态^[7]

哪些状态应该聚在一起? 使用什么模型聚类?

针对某个音素的某种状态的所有三音素状态,要聚成 C 类,哪些类聚在一起能使得似然最大,这是一个NP难问题。目前最流行的做法是采用树模型来做贪心近似聚类,如图14-17所示,树结点分裂的特征都是领域专家事先选好的,例如“右边是否为中心辅音”“左边是否是鼻音”“左边是否为摩擦音”等。最开始时所有的状态都位于根结点中,然后根据提前设计好的专家领域问题,选一个使得似然增加最大的特征进行分裂,把状态分到左、右两个结点中,后面一直递归执行,直到满足一定的结束条件为止。

图 14-17 基于决策树的状态聚类^[7]

假设做状态聚类时不改变数据对齐、转移概率，因为在做状态聚类之前，先使用单高斯模型对数据做了软对齐，这时候似然函数可以近似为如下交叉熵模型（其实就是 EM 算法的 M-Step，公式 (14.14) 去掉转移概率项）：

$$L(S) = \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma(x_{ts}^r) \log p(o_t^r | x_{ts}^r) \quad (14.74)$$

其中， S 就是需要聚类的状态集合。由于发射概率 $p(o_t | x_{ts})$ 为单高斯，因此可以推导成如下简单形式：

$$\begin{aligned} L(S) &= \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma(x_{ts}^r) \log p(o_t^r | x_{ts}^r) \\ &= \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts}) \left(-0.5 \log((2\pi)^n |\Sigma_S|) - 0.5(o_t^r - \mu_S)^T \Sigma_S^{-1} (o_t^r - \mu_S) \right) \end{aligned} \quad (14.75)$$

$$\begin{aligned}
&= -0.5 \log((2\pi)^n |\Sigma_S|) \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts}) - 0.5 \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \text{Trace}(\gamma_r(x_{ts}) \Sigma_S^{-1} (o_t^r - \mu_S)(o_t^r - \mu_S)^T) \\
&= -0.5 \log((2\pi)^n |\Sigma_S|) \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts}) - 0.5 \text{Trace}(\Sigma_S^{-1} \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts}) (o_t^r - \mu_S)(o_t^r - \mu_S)^T) \\
&= -0.5 \log((2\pi)^n |\Sigma_S|) \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts}) - 0.5 \text{Trace}(\Sigma_S^{-1} \Sigma_S \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts})) \\
&= -0.5 \left(\log((2\pi)^n |\Sigma_S|) + n \right) \sum_{r=1}^R \sum_{t=1}^T \sum_{s \in S} \gamma_r(x_{ts})
\end{aligned}$$

发射概率 $p(o_t|x_{ts})$ 为高斯函数，取对数之后就变成了二次函数，所以基于式 (14.74) 的决策树划分，等价于带权的回归树划分。因此能推导成式 (14.75) 的简单形式，最终似然只与 $\Sigma_S, \gamma_r(x_{ts})$ 相关，其中 Σ_S 可以通过统计均值和方差来直接计算， $\gamma_r(x_{ts})$ 占有后验概率可以在 Baum-Welch 参数估计时保存下来。

整个构建绑定状态 HMM 的主要步骤如下：

- (1) 使用单音素 HMM，单高斯模型初始化训练。
- (2) 三音素 HMM 直接克隆以上单音素 HMM 学到参数模型做初始化，接着训练模型，最后保存 $\gamma_r(x_{ts})$ 。
- (3) 对所有音素的所有状态做聚类。
- (4) 把单高斯扩展到混合高斯，接着训练模型。

14.8 判别式训练

前面介绍的语音识别模型 HMM 是基于最大似然 (ML) 准则来优化模型参数的，所得到的参数使得观察数据生成的概率最大。准确来说是对联合概率 $p(O, w)$ 建模，而先验概率 (语言模型) $p(w)$ 可以直接从文本数据中统计出来，不需要参加训练，因此只需要 HMM 来为似然概率 $p(O|w)$ 建模。

$$\theta = \arg \min_{\theta} - \sum_{r=1}^R \ln P(O|w_r) \quad (14.76)$$

但是语音识别的目的不是为了生成观察数据，而是要根据观察数据解码出最优的文本序列。最大似然准则要成为“最好”的训练准则，前提条件是有充足的训练数据和正确的模型。

而语音标准数据很难获取，HMM 模型假设太强，所以最大似然准则在语音识别中可能不是一个很好的训练准则。

判别式模型在机器学习有着广泛的应用，例如 Logistic Regression、支持向量机、条件随机场、最大熵马尔可夫模型等。相比于生成模型，它直接为类别的后验概率建模，往往能得到更好的效果。在语音识别等序列识别场景中，无论是 GMM-HMM、DNN-HMM 还是其他模型，使用判别式训练一般都能带来不错的效果提升。这里我们介绍三类常见的用于 HMM 判别式序列训练的目标函数：最大化互信息（MMI）、最小化分类错误（MCE）、最小化词/音素错误（MWE/MPE）。

1. 最大化互信息（MMI）

在概率统计里，互信息（Mutual Information）表示两个变量的相关性，值越大表示越相关。

$$\begin{aligned}
 I(O, W) &= \sum_{O, W} p(O, W) \log \frac{p(O, W)}{p(O)p(W)} \\
 &= \sum_{O, W} p(O, W) \log \frac{p(W|O)}{p(W)} \\
 &= H(W) - H(W|O)
 \end{aligned} \tag{14.77}$$

其中， $H(W) = -\sum_W p(W) \log p(W)$ 是文本 W 的熵， $H(W|O) = -\sum_{W, O} p(O, W) \log p(W|O)$ 是条件熵。由于 $H(W)$ 与模型参数无关，所以最大化互信息式 (14.77) 等价于最小化条件熵 $H(W|O)$ 。由于在条件熵中 $p(W|O)$ 是未知的，因此需要根据训练数据用经验条件熵来做近似：

$$H(W|O) \approx -\sum_{r=1}^R \log p(W_r|O_r) \tag{14.78}$$

利用贝叶斯公式，最小化经验条件熵等价于最大化正确序列 W_r 的后验概率：

$$O_{\text{MMI}}(\theta) = \sum_{r=1}^R \log \frac{p(O_r|W_r, \theta)^\kappa p(W_r)}{\sum_W p(O_r|W, \theta)^\kappa p(W)} \tag{14.79}$$

声学模型 $p(O_r|W_r, \theta)$ 和语言模型 $p(W_r)$ 由于模型的原因，可能在数量级上不一致，因此一般最大化互信息需要通过声学缩放因子 κ 来权衡声学模型与语言模型之间的重要性。在 $O_{\text{MMI}}(\theta)$ 优化式子中，分母需要遍历所有可能的序列，包括正确的序列和错误的序列。而在实际中不可能考虑所有的序列，只需要用概率最大的 N 个序列来参与计算做近似即可，这

N 个概率最大的序列可以通过词图 (Lattice) 来获得。

2. 最小化分类错误 (MCE)

MCE 最早是解决多分类问题的, 在孤立词识别场景中用来最小化平滑错误率, 后来才引进来用来最小化句子级别的平滑错误率, 其中的误分类测度定义为:

$$d_r(O_r) = -g_r(O_r|\theta) + G_r(O_r|\theta) \quad (14.80)$$

其中, $g_r(O_r|\theta) = \log p^\eta(O_r|w_r, \theta)$ 表示给定正确标准序列的条件对数似然概率 (η 是缩放因子), $G_r(O|\theta)$ 用来衡量非正确竞争标注序列的分数, 有很多种不同的表达形式, 最常见的为:

$$G_r(O_r|\theta) = \log \sum_{i=1, W_i \neq W_r}^N p^\eta(O_r|W_i, \theta) \quad (14.81)$$

其中, N 表示最具混淆竞争的序列数量, 这 N 个竞争序列在参数更新时都会发生变化。在实践中, 竞争序列会根据上次参数更新结果构建出来的 N -best 解码序列来确定, 用压缩形式更好的词图效率会更高。

当 $d_r(O_r) \geq 0$ 时, 表示误分类; 相反, 当 $d_r(O_r) < 0$ 时, 表示正确分类。现在我们可以通过 Sigmoid 函数来定义 MCE 损失函数:

$$l_r(d_r(O_r)) = \frac{1}{1 + e^{-\alpha d_r(O_r)}} \quad (14.82)$$

其中, α 是 Sigmoid 斜率参数, 是经验常数。最小化以上损失函数就是最小化分类错误。

在所有的序列上损失函数为:

$$O_{\text{MCE}} = \sum_{r=1}^R l_r(d_r(O_r)) \quad (14.83)$$

3. 最小化词/音素错误 (MWE/MPE)

MMI 优化的是所有序列级别的正确率, MCE 是优化序列级别的正确率, 而衡量语音识别最常用的指标是词错误率 (WER) 或音素错误率 (PER), 它们一般是根据识别出来的

词/音素序列与正确的标注序列的编辑距离计算出来的：

$$\text{WER}(\text{PER}) = \frac{S + D + I}{N} \quad (14.84)$$

其中, S 为替换的词/音素的数量, D 为删除的词/音素的数量, I 为插入的词/音素的数量, N 为所有的词/音素的数量。

MWE/MPE 的目标函数定义为：

$$O_{\text{MWE/MPE}} = \sum_{r=1}^R \frac{\sum_W p(O_r|W)^\kappa p(W) A(W_r, W)}{\sum_{W'} p(O_r|W')^\kappa p(W')} \quad (14.85)$$

其中, $A(W_r, W)$ 表示相对于正确的标注序列 W_r, W 中正确的词/音素的数量, 即 $A(W_r, W) = N - S - D - I$ 。 κ 是用来衡量声学模型和语音模型的权重, 当 $\kappa = 1$ 时, $O_{\text{MWE/MPE}} = \sum_{r=1}^R \sum_W p(W|O_r) A(W_r, W)$, 表示期望正确的词/音素数量。最大化期望正确的词/音素数量就等价于最小化词/音素错误。从直观上理解以上表达式, 对于后验概率越大的序列 W , $A(W_r, W)$ 需要尽量越大。

4. 参数估计

基于最大似然准则进行 HMM 参数估计时, 前面章节中介绍了一种基于 EM 迭代参数的求解算法——Baum-Welch, 然而在判别式训练时, EM 算法是不能直接使用的。那是什么原因造成的? 我们先看一下 MMI 判别准则, 可以表示成两个对数似然的表达形式:

$$\begin{aligned} O_{\text{MMI}}(\theta) &= \sum_{r=1}^R \log \frac{p(O_r|W_r, \theta)^\kappa p(W_r)}{\sum_W p(O_r|W, \theta)^\kappa p(W)} \\ &= \sum_{r=1}^R \left(\log p(O_r|W_r, \theta)^\kappa p(W_r) - \log \sum_W p(O_r|W, \theta)^\kappa p(W) \right) \end{aligned} \quad (14.86)$$

式 (14.86) 左边项就是我们之前讨论过的最大似然准则, 使用 Jensen 不等式就能得到该函数项的下界, EM 迭代就是不断优化下界的过程。然而式 (14.86) 右边项是需要最小化的项, 由于存在负号, 所以不能再像左边项那样使用 Jensen 不等式来获得下界。但是可以使用弱性辅助函数 (Weak-sense Auxiliary Function), 得到扩展 Baum-Welch 算法 (EB), 这里就不详细介绍这种算法了, 详细信息请参考文献 [9]。

EB 算法只适合在 GMM-HMM 模型中使用, 对 DNN-HMM 模型就不能使用了, 反倒是

基于梯度的判别式训练在这两种模型中都适用。下面我们推导 MMI 和 MWE/MPE 判别式训练的发射概率 $b_j^r(o_t) = p(o_t^r | x_{tj}^r)$ 的梯度，因为 MCE 跟 MMI 可以转化成如下统一形式，所以只需要推导 MMI 的梯度就可以了。

$$O(\theta) = \sum_{r=1}^R f\left(\log \frac{p(O_r | W_r, \theta)^\kappa p(W_r)}{\sum_W p(O_r | W, \theta)^\kappa p(W)}\right) \quad (14.87)$$

其中

$$f(x) = \begin{cases} x, & \text{如果判别式模型为MMI} \\ -\frac{1}{1+e^{\alpha x}}, & \text{如果判别式模型为MCE} \end{cases}$$

还有一点区别是，当判别式模型为 MMI 时，分母是包含正确标注序列 W_r 的；而当判别式模型为 MCE 时，分母不包含 W_r 。

5. MMI 梯度推导

$$\frac{\partial O_{\text{MMI}}}{\partial b_j^r(o_t)} = \frac{\partial (\log p(O_r | W_r, \theta)^\kappa p(W_r))}{\partial b_j^r(o_t)} - \frac{\partial \log (\sum_W p(O_r | W, \theta)^\kappa p(W))}{\partial b_j^r(o_t)} \quad (14.88)$$

根据公式 (14.45)，可得上式左边部分的求导公式：

$$\frac{\partial (\log p(O_r | W_r, \theta)^\kappa p(W_r))}{\partial b_j^r(o_t)} = \kappa \frac{\gamma^{\text{num}}(x_{tj}^r)}{b_j^r(o_t)} \quad (14.89)$$

其中， $\gamma^{\text{num}}(x_{tj}^r)$ 为第 r 正确序列标注条件下， t 时刻状态为 j 的后验概率（num 为分子 numerator 的缩写）。

同样，根据公式 (14.45)，可得右边部分的求导公式：

$$\begin{aligned} \frac{\partial \log (\sum_W p(O_r | W, \theta)^\kappa p(W))}{\partial b_j^r(o_t)} &= \frac{1}{\sum_W p(O_r | W, \theta)^\kappa p(W)} \frac{\partial (\sum_W p(O_r | W, \theta)^\kappa p(W))}{\partial b_j^r(o_t)} \\ &= \frac{1}{\sum_W p(O_r | W, \theta)^\kappa p(W)} \sum_W \frac{\partial p(O_r | W, \theta)^\kappa}{\partial b_j^r(o_t)} \cdot p(W) \\ &= \frac{1}{\sum_W p(O_r | W, \theta)^\kappa p(W)} \sum_W \frac{\partial p(O_r | W, \theta)}{\partial b_j^r(o_t)} \cdot \kappa \cdot p(O_r | W, \theta)^{\kappa-1} \cdot p(W) \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{\sum_W p(O_r|W, \theta)^\kappa p(W)} \sum_W \frac{p(O_r|W, \theta) \cdot \gamma(x_{tj}|W)}{b_j^r(o_t)} \cdot \kappa \cdot p(O_r|W, \theta)^{\kappa-1} \cdot p(W) \\
&= \frac{\kappa}{b_j^r(o_t)} \cdot \frac{\sum_W p(O_r|W, \theta)^\kappa p(W) \gamma(x_{tj}|W)}{\sum_W p(O_r|W, \theta)^\kappa p(W)} \\
&= \kappa \frac{\gamma^{\text{den}}(x_{tj}^r)}{b_j^r(o_t)}
\end{aligned} \tag{14.90}$$

其中 $\gamma^{\text{den}}(x_{tj}^r)$ 为第 r 训练样本 t 时刻状态为 j 的后验概率。

将式 (14.89) 和式 (14.90) 代入式 (14.88)，得到：

$$\frac{\partial O_{\text{MMI}}}{\partial b_j^r(o_t)} = \kappa \frac{(\gamma^{\text{num}}(x_{tj}^r) - \gamma^{\text{den}}(x_{tj}^r))}{b_j^r(o_t)} \tag{14.91}$$

6. MWE/MPE 梯度推导

$$\begin{aligned}
\frac{\partial O_{\text{MWE/MPE}}}{\partial b_j^r(o_t)} &= \frac{\frac{\kappa}{b_j^r(o_t)} \sum_{W, x_{tj}=1} \sum_{W'} - \frac{\kappa}{b_j^r(o_t)} \sum_{W', x_{tj}=1} \sum_W}{\sum_{W'} \cdot \sum_{W'}} \\
&= \frac{\kappa}{b_j^r(o_t)} \left(\frac{\sum_{W, x_{tj}=1}}{\sum_{W'}} - \frac{\sum_{W', x_{tj}=1}}{\sum_{W'}} \frac{\sum_W}{\sum_{W'}} \right) \\
&= \frac{\kappa}{b_j^r(o_t)} \frac{\sum_{W', x_{tj}=1}}{\sum_{W'}} \left(\frac{\sum_{W, x_{tj}=1}}{\sum_{W', x_{tj}=1}} - \frac{\sum_W}{\sum_{W'}} \right) \\
&= \frac{\kappa}{b_j^r(o_t)} \gamma^{\text{den}}(x_{tj}^r) (\bar{A}_r(x_{tj} = 1) - \bar{A}_r)
\end{aligned} \tag{14.92}$$

为了简化推导，以上式子在不影响阅读的前提下，省略了一些求和项中的公式符号。

转移概率 a_{ij} 可做类似推导，一般在 DNN-HMM 模型^[12] 中不再需要对转移概率更新（直接使用 GMM-HMM 得到转移概率），这里不再赘述。

在 MMI、MWE/MPE 梯度公式中， $\gamma^{\text{num}}(x_{tj}^r)$, $\gamma^{\text{den}}(x_{tj}^r)$, $\bar{A}_r(x_{tj} = 1)$, \bar{A}_r 中的未知量也是 EB 算法需要的，计算这些未知量的难度有点大，理论上要考虑所有可能的序列，而序列总数量是随时间呈指数级增长的，所以暴力求解是行不通的。对于一段语音，对应的正确的以及易混淆的序列数量是有限的。也就是说，这些易混淆序列对应的概率和占主导地位，因此可以使用这些易混淆序列近似分母。而且这些易混淆序列在 Beam-Search 解码时就能得到，如果使用词图形式存储的话，内存和计算效率会更高。

词图是在解码时使用的一种压缩存储格式，它包含了一些节点和边，其中节点表示词/音素的开始或结束时间，边代表词/音素，还包含一些其他信息，如图 14-18 所示。MMI、MWE/MPE 梯度公式中的未知量可以使用类似于 14.2.6 节介绍的前向-后向算法进行高效计算，后面第 16 章我们会介绍 Povey 博士提出的 Lattice free 判别式训练算法，基于词图的判别式训练算法会逐渐被淘汰，这里不做详细介绍，需要了解细节的读者可以参考文献 [9][10]。

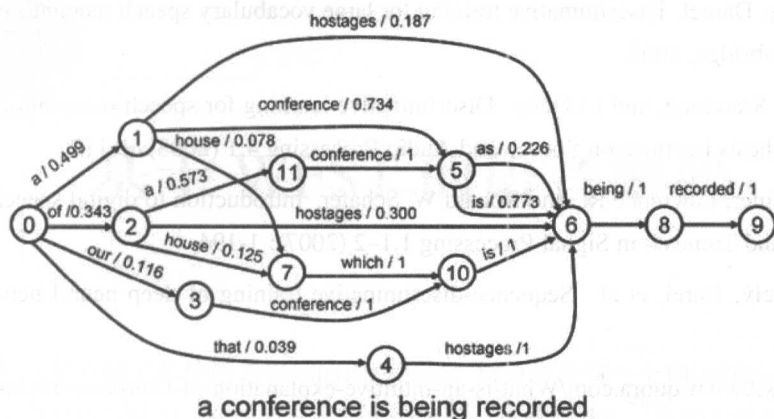


图 14-18 词图

参考文献

- [1] Bishop, Christopher M. Pattern recognition and machine learning. springer, 2006.
- [2] Boyd, Stephen, and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004.
- [3] Rabiner, Lawrence R. A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77.2 (1989): 257-286.
- [4] Cappé, Olivier, Vincent Buchoux, and Eric Moulines. Quasi-Newton method for maximum likelihood estimation of hidden Markov models. Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on. Vol. 4. IEEE, 1998.
- [5] Khreich, Wael, et al. A survey of techniques for incremental learning of HMM parameters. Information Sciences 197 (2012): 105-130.
- [6] Young, Steve, et al. The HTK book. Cambridge university engineering department 3 (2002): 175.

- [7] Gales, Mark, and Steve Young. The application of hidden Markov models in speech recognition. *Foundations and trends in signal processing* 1.3 (2008): 195-304.
- [8] Young, Steve J., Julian J. Odell, and Philip C. Woodland. Tree-based state tying for high accuracy acoustic modelling. *Proceedings of the workshop on Human Language Technology. Association for Computational Linguistics*, 1994.
- [9] Povey, Daniel. Discriminative training for large vocabulary speech recognition. Diss. University of Cambridge, 2005.
- [10] He, Xiaodong, and Li Deng. Discriminative learning for speech recognition: theory and practice. *Synthesis Lectures on Speech and Audio Processing* 4.1 (2008): 1-112.
- [11] Rabiner, Lawrence R., and Ronald W. Schafer. Introduction to digital speech processing. *Foundations and Trends® in Signal Processing* 1.1–2 (2007): 1-194.
- [12] Veselý, Karel, et al. Sequence-discriminative training of deep neural networks. *Inter-speech*. 2013.
- [13] <https://www.quora.com/What-is-an-intuitive-explanation-of-Gaussian-mixture-models>.
- [14] <http://what-when-how.com/video-search-engines/audio-query-and-browsing-techniques-audio-processing-video-search-engines/>.

基于 WFST 的语音解码

前面我们主要介绍了传统语音识别的训练部分，假设现在有一个训练好的 GMM-HMM 模型、发音字典和语言模型，你如何写出一个高效的语音解码器？一般的思路就是按部就班地实现每个流程，先根据每帧特征得到 HMM 共享状态，然后把 HMM 共享状态序列转换成音素，接下来根据发音字典（Lexicon）把多个音素合成词，再根据语言模型来决定下一个词，最后得到解码后的词序列。整个解码过程是联动的，每一帧的输入都会影响解码的每一个流程，纯手工写这个解码流程的代码特别复杂，如果要在其中加入、修改、删除子流程，或者实现一个新语言语音解码器，那么整个解码流程的代码都要重写。但是，我们发现语音解码子流程有两个明显的特点。

（1）每一个子流程的工作就是把一种串序列转化成另一种串序列，因此可以使用一种通用的工具或模块来实现。

（2）前一个子流程的输出是后一个子流程的输入，如果能把中间子流程的输出、输入约掉，只保留第一个子流程的输入和最后一个子流程的输出，形成一个直接从特征到文字序列的转换过程，那么整个解码就变得非常简单了。

带权有限状态转换器（Weighted Finite-State Transducer, WFST）就是一种能够实现以上两个特点的理想工具，接下来我们会对 WFST 的原理及其在语音识别中的应用进行详细介绍。

15.1 有限状态机

有限状态机 (Finite Automata, FA) 是表示有限个状态以及在这些状态之间转移和动作等行为的数学模型, 在不同的领域有着广泛的应用。根据功能的不同, 有限状态机又可以分为有限状态接收器 (FSA) 和有限状态转换器 (FST)。给定输入序列, FSA 返回“是”或“否”来表示当前输入序列被机器接受还是拒绝, 当所有输入都处理了, 而且当前状态是结束状态时, 表示该输入序列被接受; 否则被拒绝。如果想知道被接受的概率或权重, 那么 FSA 就变成了带权有限状态接收器 (WFSA)。给定输入序列, FST 返回的是输出序列, 每个转移接收一个标签, 输出一个标签。同样, 如果想知道每种输出的概率或权重, 那么 FST 就变成了带权有限状态转换器 (WFST)。其实 FSA、WFSA、FST 都是 WFST 的特殊情况, 如图 15-1 所示。

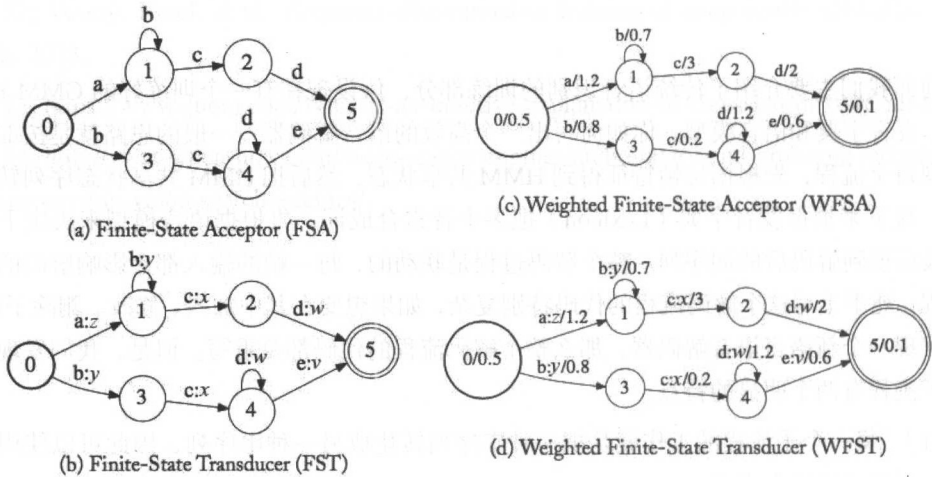


图 15-1 4 种不同的有限状态机^[1]

15.2 WFST 及半环定义

15.2.1 WFST

WFST 是定义在权重集合 \mathbb{k} 上的 8 元组—— $(\Sigma, \Delta, Q, I, F, E, \lambda, \rho)$, 其中:

- Σ 是有限的输入标签集合;

- Δ 是有限的输出标签集合;
- Q 是有限状态集合;
- $I \in Q$ 是初始状态集合;
- $F \in Q$ 是结束状态集合;
- $E \in Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \mathbb{k} \times Q$ 是有限转移集合, 其中 ϵ 表示没有输入或输出的元符号标签;
- $\lambda: I \rightarrow \mathbb{k}$ 初始状态权重函数;
- $\rho: F \rightarrow \mathbb{k}$ 结束状态权重函数。

15.2.2 半环 (Semiring)

在带权有限自动机里, 权重及其相关的二元操作“加法”和“乘法”用来形式化定义与自动机相关的算法, 这里所涉及的“加法”和“乘法”都是广义上的。自动机有一个很重要的性质, 就是: 如果给定的权重及其二元操作定义了一个半环, 那么权重集合中的所有权重都可以被自动机处理。除加法逆运算不满足之外, 半环跟环非常相似, 半环定义为一个 5 元组—— $(\mathbb{k}, \oplus, \otimes, \bar{0}, \bar{1})$, 其中:

- \mathbb{k} 是元素集合;
- \oplus 是广义上的加法;
- \otimes 是广义上的乘法;
- $\bar{0}$ 是加法运算的单位元;
- $\bar{1}$ 是乘法运算的单位元。

半环必须满足以下公理性质。

- 加法满足结合律: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- 加法满足交换律: $x \oplus y = y \oplus x$
- 乘法满足结合律: $(x \otimes y) \otimes z = x \otimes (y \otimes z)$
- 满足分配律: $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$, $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$
- 加法单位元性质: $\bar{0} \oplus x = x \oplus \bar{0} = x$, $\bar{0} \otimes x = x \otimes \bar{0} = \bar{0}$
- 乘法单位元性质: $\bar{1} \otimes x = x \otimes \bar{1} = x$

表 15-1 给出了 5 种基于 WFST 的半环。

表 15-1 5 种 WFST 半环定义^[6]

Semiring	Set	\oplus	\otimes	$\bar{0}$	$\bar{1}$
Boolean	$\{0, 1\}$	\vee	\wedge	0	1
Probability	\mathbb{R}_+	+	\times	0	1
Log	$\mathbb{R} \cup \{-\infty, +\infty\}$	\oplus_{\log}	+	$+\infty$	0
Tropical	$\mathbb{R} \cup \{-\infty, +\infty\}$	min	+	$+\infty$	0
String	$\Sigma^* \cup \{\infty\}$	\wedge	\cdot	∞	ϵ

其中, $x \oplus_{\log} y = -\log(e^{-x} + e^{-y})$, 跟热带 (Tropical) 半环有点类似, 得到的值近似于取最小值。 \wedge 表示字符串最长公共前缀。在基于 WFST 的语音识别系统中, 解码时最常用的就是热带半环和对数半环, 其中的加法操作就是在多条路径中取权值最小的路径, 乘法操作就是负对数概率值相加。

以上介绍了半环必须满足的条件。下面列举一些其他重要的性质, 满足这些性质的半环可以简化一些算法计算, 而且有的 WFST 操作必须满足这些性质。

1. 可交换 (Commutative)

如果半环 $(\mathbb{k}, \oplus, \otimes, \bar{0}, \bar{1})$ 的乘法操作满足交换律, 即对于任意的 $x, y \in \mathbb{k}$, 满足 $x \otimes y = y \otimes x$, 那么该半环是可交换的。热带半环和对数半环都是可交换的。

2. 幂等 (Idempotent)

如果半环 $(\mathbb{k}, \oplus, \otimes, \bar{0}, \bar{1})$ 满足对于任意的 $x \in \mathbb{k}$, 都有 $x = x \oplus x$, 那么该半环是幂等的。热带半环是幂等的, 但对数半环不是。

3. k -闭包 (k -Closed)

对于 $k \geq 0$ 的整数, 如果半环 $(\mathbb{k}, \oplus, \otimes, \bar{0}, \bar{1})$ 满足对于任意的 $x \in \mathbb{k}$, 都有:

$$\bigoplus_{n=0}^{k+1} x^n = \bigoplus_{n=0}^k x^n$$

那么该半环是 k -闭包的。布尔 (Boolean) 半环和热带半环是 0-闭包的。

4. 弱左可除 (Weakly left-divisible)

如果半环 $(\mathbb{k}, \oplus, \otimes, \bar{0}, \bar{1})$ 满足对于任意的 $x, y \in \mathbb{k}$ 且 $x \oplus y \neq \bar{0}$, 至少存在一个 z 使得 $x = (x \oplus y) \otimes z$, 那么该半环是弱左可除的。热带半环和对数半环都是弱左可除的。

5. 无零和 (Zero-sum free)

对于任意的 $x, y \in \mathbb{k}$, 如果 $x \oplus y = 0$, 能推断出 $x = y = \bar{0}$, 那么该半环是无零和的。

15.3 自动机操作

在自动机理论中, 有很多一元、二元基本操作, 使用这些操作可以完成: 生成新的更加复杂的自动机、层级组合自动机、简化自动机、确定化自动机和优化自动机等, 很多经典的自动机操作只需要少量的修改, 就可以直接应用到 WFST 中。

下面先简单介绍几种基本操作, 后面再详细介绍复合 (Composition) 操作及一些优化操作, 这些操作在语音识别系统的 WFST 中特别重要。

在详细介绍这些操作之前, 先给出一些基本术语, 便于描述后续的算法。

- $E[q]$: 表示从状态 $q \in Q$ 发出的所有转移的集合;
- $i[e]$: 表示转移 $e \in E$ 的输入标签;
- $o[e]$: 表示转移 $e \in E$ 的输出标签;
- $p[e]$: 表示转移 $e \in E$ 的源状态;
- $n[e]$: 表示转移 $e \in E$ 的目的状态;
- $w[e]$: 表示转移 $e \in E$ 的权重;
- $\pi = e_1, \dots, e_k$: 满足 $n[e_{j-1}] = p[e_j], j = 2, \dots, k$, 表示一系列连续转移的路径, 长度为 k ;
- $n[\pi]$: 表示路径的目的状态, $n[\pi] = n[e_k]$;
- $p[\pi]$: 表示路径的源状态, $p[\pi] = p[e_1]$;
- $o[\pi] = o[e_1] \cdots o[e_k]$: 表示路径的输出;
- $i[\pi] = i[e_1] \cdots i[e_k]$: 表示路径的输入;
- $w[\pi] = w[e_1] \otimes \cdots \otimes w[e_k]$: 表示路径的权重。

15.3.1 自动机基本操作

1. 克林闭包 (Kleene Closure)

克林闭包修改自动机可以使其中的符号序列重复 0 次或多次, 自动机 A 的克林闭包记为 A^* , 正则表达式中的 $*$ 运算符可以使用克林闭包实现, 如图 15-2 (c) 所示。

2. 并操作 (Union)

并操作把两个自动机合并为一个新的自动机, 新的自动机能够接受或转换的序列是之前两个自动机的并集, 如图 15-2 (d) 所示。

3. 串联 (Concatenation)

串联操作把其中一个自动机接在另一个自动机的结束状态后面, 串联得到新的自动机, 如图 15-2 (e) 所示。

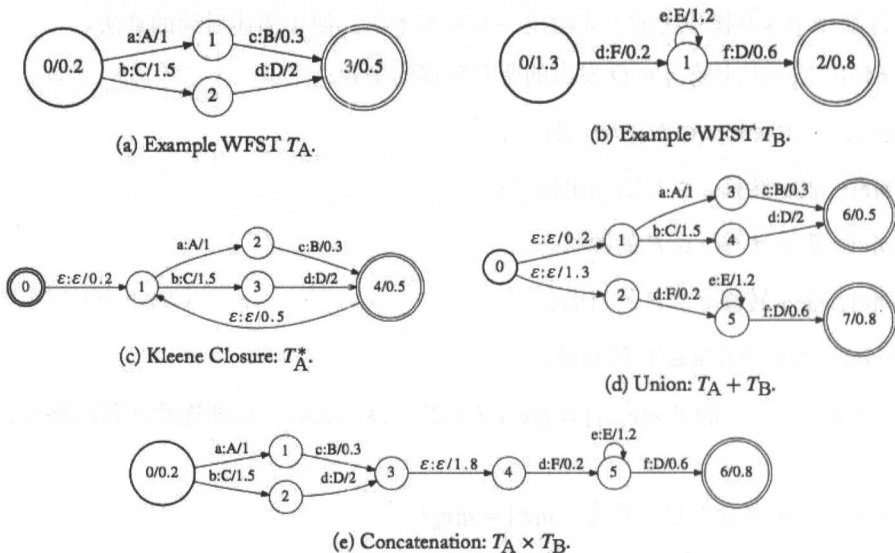


图 15-2 三种自动机基本操作^[1]

15.3.2 转换器基本操作

1. 投影 (Projection)

投影操作忽略输入或输出，把转换器变成接收器。

2. 倒置 (Inversion)

倒置操作把转换器的每个转移中的输入和输出符号倒置过来。

3. 组合 (Composition)

组合操作把两个转换器合并成一种级联的转换器。如图 15-3 所示的是一个字母转大写的转换器 T_a ，输入的是字母序列，输出的是大写字母序列。如图 15-4 所示的是一种从大写字母序列中挑选 RED, BLUE, GREEN 单词的转换器 T_b ，输入的是大写字母序列，输出的是挑选后的单词。如果要想实现一个从字母序列直接挑选单词的功能，则需要先用 T_a 把字母序列转换输出为大写字母序列，该序列作为 T_b 的输入，最终实现挑选单词的功能，这样做显得特别麻烦、冗余。而通过组合操作能把多个转换器组合成一个转换器，如图 15-5 所示是 T_a 和 T_b 组合成新的转换器 T_c ，这样就可以实现直接从字母序列挑选单词的功能了。

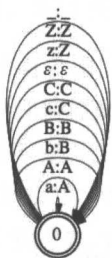


图 15-3 字母大写转换器 T_a ^[1]

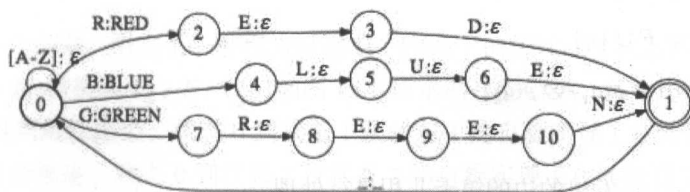
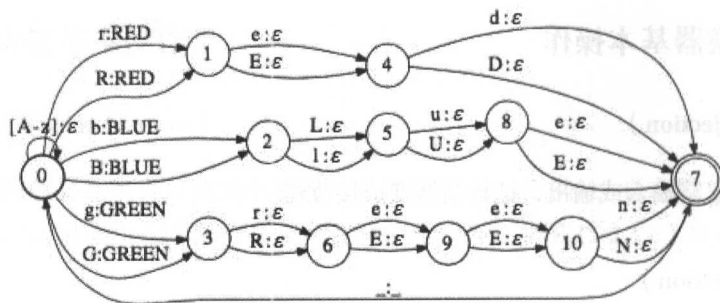


图 15-4 单词挑选器 T_b ^[1]

图 15-5 $T_a \circ T_b^{[1]}$

不是所有的转换器之间都能进行组合操作，必须要满足其中一个转换器的输出是另一个转换器的输入。下面给出了 $T_1 = (\Sigma_1, \Delta_1, Q_1, I_1, F_1, E_1, \lambda_1, \rho_1)$ 和 $T_2 = (\Sigma_2, \Delta_2, Q_2, I_2, F_2, E_2, \lambda_2, \rho_2)$ 两个转换器之间的组合算法。

算法 15-1 转换器组合算法

// T_1, T_2 构建组合初始状态，针对两个状态的笛卡儿积生成初始状态组合

for each $(i_1, i_2) \in I_1 \times I_2$ do

$\lambda((i_1, i_2)) = \lambda_1(i_1) \otimes \lambda_2(i_2)$ // 组合初始状态权重函数

$I = I \cup \{(i_1, i_2)\}$ // 组合初始状态

end for

$Q = I$ // 初始化组合状态集合等于 I

$S = I$ // 任务队列初始化为 I

// 广度优先生成组合转换器

while $S \neq \emptyset$

$q = (q_1, q_2) = \text{Dequeue}(S)$ // 出队列

// 如果是结束状态，则需要计算组合结束状态权重函数

if $q \in F_1 \times F_2$ then

$F = F \cup \{q\}$

$\rho((q)) = \rho(q_1) \otimes \rho(q_2)$

end if

// 对于 q_1, q_2 发出的边的笛卡儿积进行处理

for each $(e_1, e_2) \in E[q_1] \times E[q_2]$, 且需要满足 $o[e_1] = i[e_2]$ do

```

// 有些组合状态可能之前就生成了，不再做重复处理

if  $q' = (n[e_1], n[e_2]) \notin Q$  then
     $Q = Q \cup q'$  // 加入新的状态
    Enqueue( $S, q'$ ) // 入队列
end if

// 加入新的转移边

 $E = E \cup \{(q, i[e_1], o[e_2], w[e_1] \otimes w[e_2], q')\}$ 

end for

end while

return  $T = (\Sigma_1, \Delta_2, Q, I, F, E, \lambda, \rho)$ 

```

以上组合算法从生成组合初始状态 (i_1, i_2) 开始，使用广度优先遍历不断地生成新的合法的组合状态 (q_1, q_2) ，新的组合权重为 $w[e_1] \otimes w[e_2]$ ，最终的组合转换器的输入为 T_1 的输入 Σ_1 ，输出为 T_2 的输出 Δ_2 。在最坏的情况下，在 T_1 中所有从状态 q_1 出发的转移都跟 T_2 中所有从状态 q_2 出发的转移匹配，那么复杂度为 $O(|T_1||T_2|)$ ，其中 $|T_i| = |Q_i| + |E_i|$ 。但是在一般的场景中，复杂度是远低于最坏情况的，所以在实际应用中该算法还是特别快的。

以上介绍的组合算法没有考虑当 T_1 中有 ε 输出、 T_2 中有 ε 输入时的情况，当 T_1 中有 ε 输出时， T_1 可以做一次 ε 转移，而这时 T_2 保持原来的状态不变；相反，当 T_2 中有 ε 输入时， T_2 可以做一次 ε 转移， T_1 保持原来的状态不变。当遇到这两种情况时，不能直接把 ε 当作正常的字符处理，否则会使得 T_1 的 ε 转移跟 T_2 的 ε 转移做组合，这显然是错误的。一般的做法如图 15-6 所示，把图 (a) 和图 (b) 所示的 T_1, T_2 扩展成图 (c) 和图 (d) 所示的 T'_1, T'_2 ，把 $\varepsilon o, \varepsilon i$ 当成正常字符处理，然后直接对 T'_1, T'_2 做组合，得到的组合转换器如图 15-7 所示。这种做法存在两个问题：其一，从状态 (1, 1) 到状态 (2, 3) 存在冗余等价的路径；其二，如果 WFST 不是幂等的，结果就会出现错误，例如概率半环和对数半环。

为了解决以上问题，引入了一种称为过滤器 (Filter) 的特殊的 3 状态转换器 F ，将原来的组合操作 $T'_1 \circ T'_2$ 替换成 $T'_1 \circ F \circ T'_2$ 。如图 15-8 所示的过滤器 F ，其中 $x \in \Sigma_2 \cap \Delta_1$ ，转移 $\varepsilon o : \varepsilon i$ 是为了得到最短路径，状态 0 到状态 1 及状态 1 的自旋实现了在转换器 T_1 连续的 ε 输出方向上的组合推进，状态 0 到状态 2 及状态 2 的自旋实现了在转换器 T_2 连续的 ε 输入方向上的组合推进，其中的自旋逻辑消除了冗余路径。最终，使用过滤器的组合操作结果如图 15-9 所示。

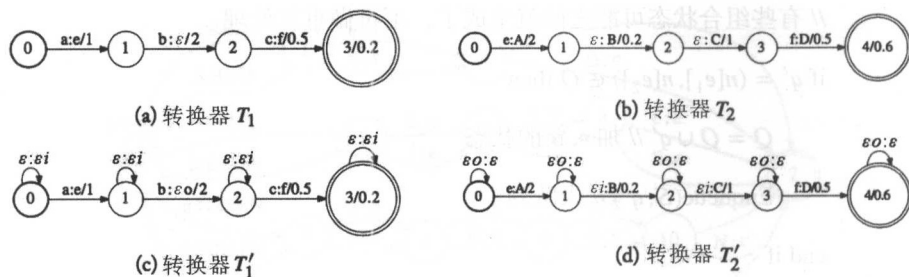


图 15-6 转换器扩展^[1]

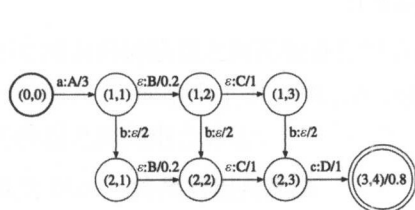


图 15-7 $T_1' \otimes T_2'^{[1]}$

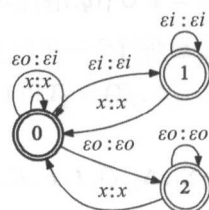


图 15-8 3 状态过滤器 $F^{[1]}$

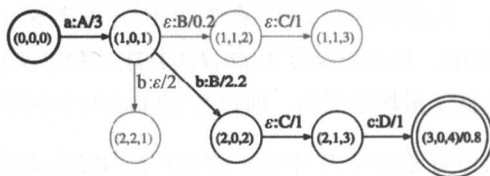
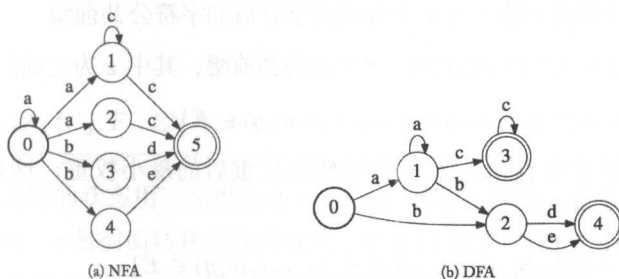


图 15-9 $T_1' \circ F \circ T_2'^{[1]}$

15.3.3 优化操作

1. 确定化

有限状态机分为确定有限状态机 (Deterministic Finite Automata, DFA) 和非确定有限状态机 (Non-deterministic Finite Automata, NFA)。如果有限状态机满足只有单个的初始状态, 而且每个状态对于任意的输入标签有且仅有一个转移边, 那么该有限状态机就是 DFA, 反之就是 NFA, 如图 15-10 所示。对于 DFA 来说, 给定一个可接受的输入序列, 从初始状态到结束状态只有一条路径与之对应。

图 15-10 DFA 和 NFA^[1]

DFA 相比于 NFA 有一个最大的优势, 就是计算非常快。给定一个序列, 判断其是否可接受, NFA 在每次转移时都需要考虑多条路径, 在最坏的情况下复杂度为 $O(L|Q||E|)$, 其中 L 表示序列的长度; 而 DFA 在每次转移时可以用二分查找, 所以时间复杂度为 $O(L \log \hat{D})$, 其中 \hat{D} 为状态中最大的出度。幸运的是, 存在把 NFA 转换成与之等价的 DFA 的算法, 称为确定化 (Determinization) 算法。

将传统的针对有限状态机的确定化算法稍加修改就能得到 WFST 的确定化算法, 能做确定化的 WFST 必须是满足弱左可除的半环。下面给出了 WFST 的确定化算法, 为了给出代码的详细解释, 假设当前处理的 WFST 为热带半环。

算法 15-2 WFST 的确定化算法

// 初始状态前面的输出残差为空 ε , 权重残差为原来的初始状态权重函数

$i' = \{(i, \varepsilon, \lambda(i)) | i \in I\}$

$\lambda'(i') = \bar{1}$

// 确定化的初始状态初始化为三元组

$Q' = \{i'\}$

// 采用广度优先遍历做确定化

$S = \{i'\}$

while $S \neq \emptyset$ do

$p' = \text{Dequeue}(S)$

 // 对于三元组集合 p' 所涉及的所有可能的输入, 其中特定的输入 x 对应的边可能有多条

 // 这种情况就需要做确定化, 把多条输入相同的边合为一条

 for each $x \in \{x | i[e] = x, e \in E[p], p \in Q[p']\}$ do

// 计算所有输入为 x 合并残差字符后的字符公共前缀

// 其中 “.” 是字符拼接, “~” 是公共前缀, 其中 z 为之前的字符残差

$$y' = \wedge \{z.y | (p, z, v) \in p', (p, x, y, w, q) \in E\}$$

// 计算所有输入为 x 合并残差权重后的最小权重, 这是为了保证残差权重 $q' \geq 0$ (假设为热带半环)

$$w' = \oplus \{v \otimes w | (p, z, v) \in p', (p, x, y, w, q) \in E\}$$

// 计算新的确定化的状态三元组集合, 其中每个三元组都包含原始状态 q

// 残差字符序列为 $y'^{-1}.z.y$, 表示把公共前缀 y' 从 $z.y$ 中减去了

// 残差权重为 $\oplus \{w'^{-1} \otimes v \otimes w | (p, z, v) \in p', (p, x, y, w, q) \in q\}$, 前一次的残差权重 v 加上本身的权重 w , 再减去最小权重, 然后取最小权重

$$q' = \{(q, y'^{-1}.z.y, \oplus \{w'^{-1} \otimes v \otimes w | (p, z, v) \in p', (p, x, y, w, q) \in q\}) | (p, z, v) \in p', (p, x, y, w, q) \in E\}$$

// 加入新的边

$$E' = E' \cup \{(p', x, y', w', q')\}$$

// 可能会生成重复的状态

if $q' \notin Q'$ then

// 加入新的状态

$$Q' = Q' \cup \{q'\}$$

// 处理结束状态

if $Q[q'] \cap F \neq \emptyset$ then

// 加入终止状态

$$F' = F' \cup \{q'\}$$

// 结束状态的权重函数是一个二元组集合, z 为字符残差, $\oplus \{v \otimes \rho(q)\}$

只保留了最小权重

$$\rho'(q') = \{(z, \oplus \{v \otimes \rho(q) | (q, z, v) \in q', q \in F\}) | (q, z, v) \in q', q \in F\}$$

end if

// 加入队列进行下一次处理

Enqueue(S, q')

end if

end for

end while

return $T' = (\Sigma, \Delta, Q', \{i'\}, F', E', \lambda', \rho')$

以上算法确定化之后的状态用三元组集合 $\{(p, z, v)\}$ 表示, 其中 p 是确定化前的状态, z 是前面转移的残差输出, v 是前面转移的残差权重。其中的 $Q[p']$ 操作是把三元组 p' 中的状态信息提取出来。如图 15-11 (a) 所示是非确定化的 WFST, 如图 15-11 (b) 所示是使用以上确定化算法得到的一种称为 p -subsequential 的转换器, 能很容易把 p -subsequential 转换成 WFST, 最终得到如图 15-11 (c) 所示的 WFST。

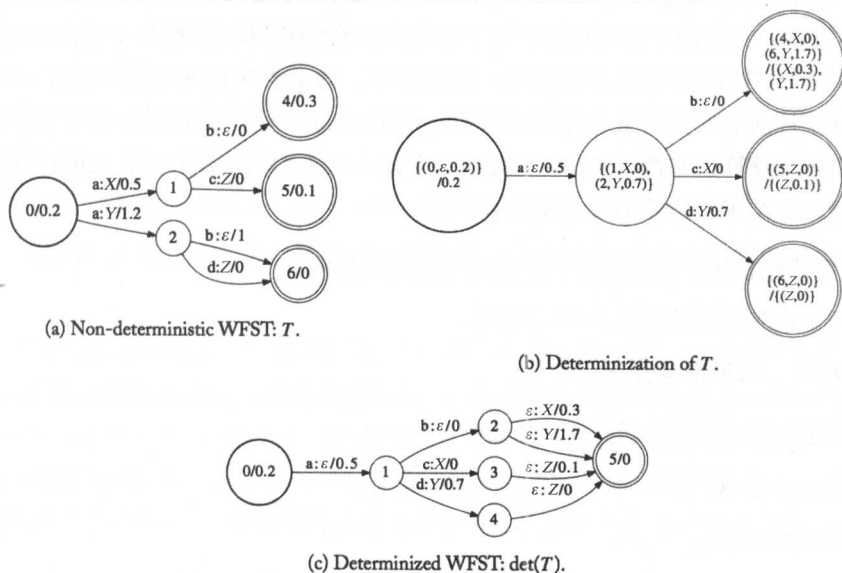
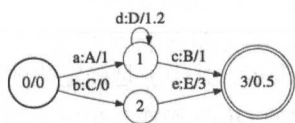
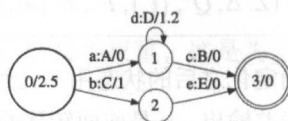


图 15-11 WFST 确定化^[1]

2. 权重推移

对于 WFST 上面的一条成功路径, 我们通常只在乎该路径的总权重大小, 而不关心总权重在路径上的每个转移的权值分布。如果把权重尽可能地往初始状态方向移动, 那么对于某个时刻的状态点, 我们就能提前知道整条路径权重的一个下界, 就能快速地把一些不靠谱的路径提早过滤掉。例如, 在语音识别系统的 WFST 中, 我们的最终任务是寻找到一条概率最大的解码路径, 使用权重推移 (Weight Pushing) 操作, 可以提早去掉一些不必要的路径, 即如果某些路径比给定路径的下界概率都小的话, 那么应该可以提前去掉, 这样可以大大加快

解码速度。如图 15-12 所示是权重推移之前的 WFST，初始状态的权重值是 0。如图 15-13 所示是权重推移之后初始状态就能得到 2.5 的权重值，因此可以加快剪枝。

图 15-12 WFST 权重推移之前^[1]图 15-13 WFST 权重推移之后，热带半环^[1]

权重推移包含两个主要的步骤。

(1) 计算每个状态 q 到终止状态的最短路径 $d[q]$ ，如果定义在 WFST 中的半环满足 k -闭包，则可以采用广义上的单源最短路径算法来完成，算法复杂度为 $O(|E| + |Q| \log |Q|)$ 。当然，如果是有向无环图的话，算法复杂度就是线性的了。如果半环只满足是完全的 (Complete) ^[6]，则需要采用所有节点对最短路径算法 Floyd-Warshall，该算法的复杂度为 $|Q|^3$ ，在状态特别多的时候，该算法非常不实用。但是可以采用一些近似的快速算法，也能得到不错的效果，这里就不介绍了，详情请参考文献 [6]。

(2) 对初始权重、转移权重、终止权重重新赋权值。

整个权重推移的算法如算法 15-3 所示。

算法 15-3 权重推移算法

// 计算终止状态到所有状态的最短路径

for each $q \in Q$ do

 compute $d[q]$

end for

for each $q \in Q$ do

 // 重新对初始权重赋值

 if $q \in I$ then

$\lambda(q) = \lambda(q) \otimes d[q]$

 end if

 // 重新对转移权重赋值

 for each $e \in E[q]$ do

$w[e] = d[q]^{-1} \otimes w[e] \otimes d[n[e]]$

```

end for
// 重新对终止权重赋值
if  $q \in F$  then
     $\rho(q) = d[q]^{-1} \otimes \rho(q)$ 
end if
end for

```

3. 最小化

给定一个 DFA，如果不存在其他等价的 DFA，其状态数量比当前给定的 DFA 少的话，那么我们说当前给定的 DFA 是最小的 (Minimal)。最小化 (Minimization) 算法是一种把 DFA 转换成等价的、含有最少状态数量的 DFA 的操作。显然，对于等价的两个 DFA，状态数量少的那个 DFA 更受欢迎，因为基本上在 DFA 上运行的所有操作，其时间、空间复杂度都会随状态数量的增加而变复杂，因此一般在使用 DFA 前需要对其进行最小化。

同样，WFST 也可以运行最小化算法，但是在运行最小化算法之前需要执行以下两个步骤。

(1) 权重推移和输出前移。前面我们只介绍了权重推移算法，输出前移操作并没有介绍，其实如果把输出序列当成一种特殊的权重，也就是在输出字符上定义一个字符 (String) 半环 (\otimes 为字符连接操作 $.$ ， \oplus 为求公共前缀操作 \wedge)，就可以完全使用前面介绍的权重推移算法来执行输出前移操作。为什么要进行权重推移和输出前移操作呢？因为运行完推移操作之后，会使得很多输出为 ϵ ，很多权重为乘法单位元 $\bar{1}$ ，这样会加大状态合并的力度。

(2) 最小化 WFST 可以使用传统经典的 Hopcroft 算法，只需要把 $x : y/w$ 三元组当成单一的标签就行。

算法 15-4 最小化 Hopcroft 算法

```

//  $P$  为当前的划分，初始化为空
 $P = \emptyset$ 
//  $W$  是用于划分  $P$  中的块的等待队列，初始化为空
 $W = \emptyset$ 
// 把有相同权值的结束状态划分到同一个块中
for each  $\rho \in \{\rho(f) | f \in F\}$  do

```

$$F_\rho = \{f | \rho(f) = \rho, f \in F\}$$

$$P = P \cup F_\rho$$

// 把结束状态块作为引子用于将来划分 P 中的块

Enqueue(W, F_ρ)

end for

// 在划分中加入非结束状态划分块

$$P = P \cup \{Q - F\}$$

while $W \neq \emptyset$ do

$S = \text{Dequeue}(W)$

// 对于 S 中所有入边可能的 (i, o, w) 三元组进行遍历

for each $(i, o, w) \in \{(i[e], o[e], w[e]) | e \in E^{-1}[S]\}$ do

// 在 S 的所有入边 e 中, 其输入、输出、权值分别为 (i, o, w) 的初始状态 $p[e]$ 的集合, $R_{i,o,w}$

// $R_{i,o,w}$ 中的所有状态都是等价的, 需要合并

$$R_{i,o,w} = \{p[e] | i[e] = i, o[e] = o, w[e] = w, e \in E^{-1}[S]\}$$

// 对于 P 中满足条件的块 B , 需要进一步划分

for each $B \in P$, 满足 $B \cap R_{i,o,w} \neq \emptyset$ 而且 $B \not\subseteq R_{i,o,w}$ do

// 交集, B_1 里面的状态不一定是等价的状态

$$B_1 = B \cap R_{i,o,w}$$

// 差集

$$B_2 = B - B_1$$

// 将 B 划分成 B_1, B_2 两个子块

$$P = (P - \{B\}) \cup \{B_1, B_2\}$$

// 如果 B 在等待队列里面, 那么用新的子块替换原来的大块, 能得到更加精细的划分引子

if $B \in W$ then

Replace B with B_1, B_2

else // B 不在等待队列里面

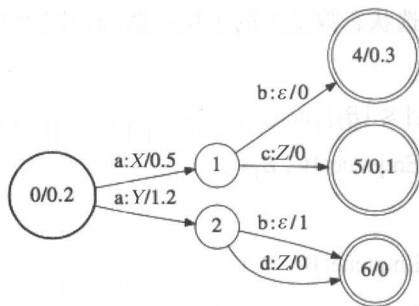
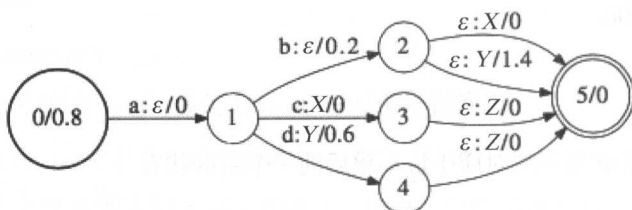
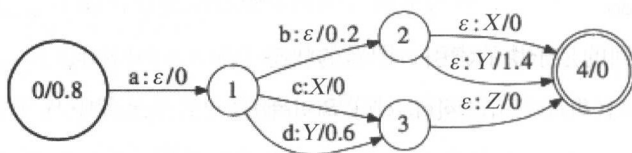
度更低

```

// 挑选状态数量少的子块入队列, 因为对于小的集合后面处理复杂度更低
if  $|B_1| \leq |B_2|$  then
    Enqueue( $W, B_1$ )
else
    Enqueue( $W, B_2$ )
end if
end if
end for
end while
// 每个划分后的状态子块对应于最小化后的一个新的状态
 $Q' = P$ 
// 生成新的边集合
for each  $e \in E$  do
    //  $B(s)$  返回的是子块的索引, 即新的状态
     $E' = E' \cup \{(B(p[e]), i[e], o[e], w[e], B(n[e]))\}$ 
end for
// 生成新的终止状态
for each  $S \in Q'$  而且  $S \subseteq F$  do
     $F' = F' \cup \{S\}$ 
     $\rho'(S) = \rho(q)$ , 其中  $q \in S$ 
end for
return  $T' = (\Sigma, \Delta, Q', I, F', E', \lambda, \rho')$ 

```

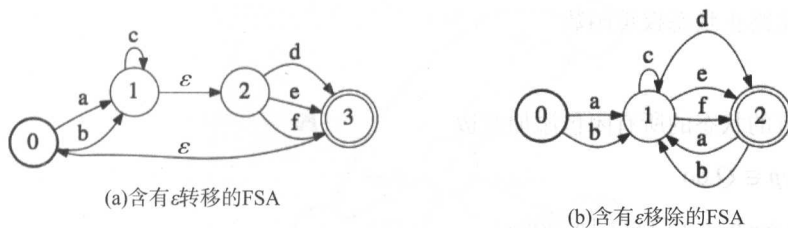
以上给出的是 WFST 最小化 Hopcroft 算法, 其时间复杂度为 $O(|E| \log |Q|)$ 。如果 WFST 是有向无环图的话, 那么可以使用 Revuz 算法, 其复杂度为 $O(E)$ 。图 15-14 给出了一个原始的 WFST, 图 15-15 给出了一个经过确定化、权重推移的 WFST, 图 15-16 给出了一个经过确定化、权重推移和最小化的 WFST, 从原始的非确定的 WFST 变成最后确定化、最小化的 WFST, 状态数量从原来的 6 个变成了 5 个, 而且还是确定化的, 因此图 15-16 所示的 WFST 相比于原始的 WFST 在性能上有了质的提升。

图 15-14 原始的 WFST, $T^{[1]}$ 图 15-15 经过确定化、权重推移的 WFST, $\text{push}(\text{det}(T))^{[1]}$ 图 15-16 经过确定化、权重推移、最小化的 WFST, $\min(\text{push}(\text{det}(T)))^{[1]}$

4. ϵ 删除

前面我们介绍了基本的非确定有限状态机，这里将介绍一种常见类型的 NFA—— ϵ 非确定有限状态机 (ϵ -NFA)，它是一种含有 ϵ 转移 (ϵ 输入) 的有限状态机。如图 15-17 (a) 所示，状态 1 与状态 2 之间有 ϵ 转移，因此状态机同一时刻可能同时处于状态 2 和状态 3，所以存在着 ϵ 输入，给状态机带来了不确定性。可以使用 ϵ 移除算法把 ϵ 移除得到等价的无 ϵ DFA，如图 15-17 (b) 所示是与图 15-17 (a) 所示等价的 DFA。

针对 NFA 的确定化算法也可以直接应用到 ϵ -NFA 上，只需要把 ϵ 当成正常的输入即可，但是确定化后的自动机还是含有 ϵ 输入的，也就是说，确定化不完全，还需要进行 ϵ 移除，才能变成完全确定化的 DFA。下面我们详细介绍 ϵ 移除优化算法。

图 15-17 ϵ 删除操作^[1]

简单来说, ϵ 移除算法就是: 删除 ϵ 转移, 添加新的非 ϵ 转移。在任意两个状态之间, 如果可以通过一个或连续多个 ϵ 转移, 最后再加上一个非 ϵ 转移可以到达的话, 那么这两个状态直接就可以新增一个非 ϵ 转移边, 然后把途径的 ϵ 转移删除。终止状态需要特殊处理: 如果一个非终止状态到终止状态只经过了 ϵ 转移, 那么该非终止状态在进行完 ϵ 移除之后就成为了新的终止状态。要找出在哪些状态之间可以添加新的转移, 首先需要对每个状态找出 ϵ -闭包, 在给定状态 p 下, 所谓的 ϵ -闭包就是一组只能通过 ϵ 转移到达的状态集合 $C(p)$, 与 $C(p)$ 通过非 ϵ 转移的状态 q 与给定的状态 p 之间就可以添加新的转移了。

由于 WFST 在做 ϵ 移除时, 不仅需要考虑输入, 还需要考虑输出和权重, 所以处理起来比 FA 要复杂得多。这里介绍的算法只考虑输入和输出都是 ϵ 的情况, 即 $\epsilon : \epsilon/w$, 对于转移输入输出中只包含一个 ϵ 的情况暂时不考虑, 即 $\epsilon : x/w, x : \epsilon/w$, 这种情况需要首先使用同步 (Synchronization) 操作来尽可能地使 $\epsilon : x/w, x : \epsilon/w$ 转移减少, $x : x/w, \epsilon : \epsilon/w$ 的情况增多, 然后再使用 ϵ 转移算法。同步算法这里就不做介绍了, 有兴趣的读者可以参考文献 [6]。下面给出了 WFST 的 ϵ 移除算法。

算法 15-5 WFST 的 ϵ 移除算法

// 对所有状态, 计算 ϵ -闭包

for each $p \in Q$ do

 计算闭包 $C(p)$

// 收集所有非 (ϵ, ϵ) 转移

$\overline{E}_\epsilon = \{(p, a, b, w, q) \in E : (a, b) \neq (\epsilon, \epsilon)\}$

// 非 (ϵ, ϵ) 转移肯定会留着

$E' = \overline{E}_\epsilon$

// 初始化终止状态集合

$F' = F$

```

// 初始化终止状态权重函数
 $\rho' = \rho$ 
// 对所有的状态的所有闭包添加新边
for each  $p \in Q$  do
    for each  $(q, w') \in C[p]$  do
        // 添加新边, 权重等于闭包的最短路径乘上非  $\epsilon$  转移的权重:  $w' \otimes w$ 
         $E'[p] = E'[p] \cup \{(p, a, b, w' \otimes w, r) : (q, a, b, w, r) \in \overline{E}_\epsilon\}$ 
        if  $q \in F$  then
            // 如果一个非终止状态到终止状态只经过了  $\epsilon$  转移,
            // 那么该非终止状态在进行完  $\epsilon$  移除之后就成为了新的终止状态
            if  $p \notin F$  then
                 $F' = F \cup \{p\}$ 
                 $\rho'[p] = \overline{0}$ 
            end if
            // 例如在热带半环中, 就是取一个最小的终止状态权重函数
             $\rho'[p] = \rho'[p] \oplus \{w' \otimes \rho(q)\}$ 
        end if
    end if
return  $T' = (\Sigma, \Delta, Q, I, F', E', \lambda, \rho')$ 

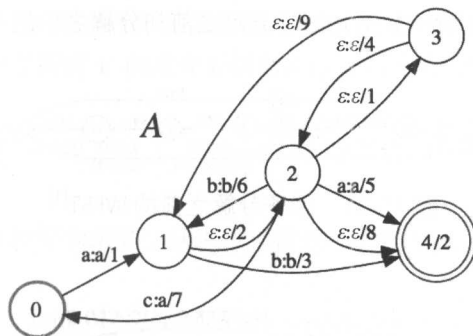
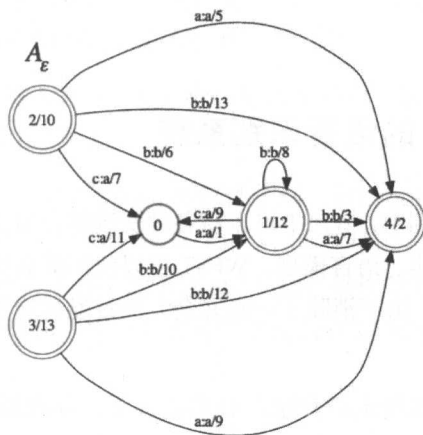
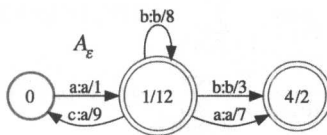
```

WFST 中的带权 ϵ -闭包定义如下:

$$C(p) = \{(q, w) : P(p, \epsilon, q) \neq \emptyset, w = d_\epsilon[p, q]\}$$

其中, $P(p, \epsilon, q)$ 是从状态 p 到 q 的 ϵ 路径, $d_\epsilon[p, q] = \bigoplus_{\pi \in P(p, \epsilon, q)} w[\pi]$ 表示最短路径上的权重。如果半环是完全的 (Complete), 那么带权 ϵ -闭包可以使用广义上的所有节点对的最短路径算法 (Floyd-Warshall 算法); 如果半环是 k -闭包的, 则可以使用广义上的单源最短路径算法 (Dijkstra 算法); 如果还是有向无环图的话, 那么可以使用基于拓扑排序的最短路径算法。

注意到以上 ϵ 移除算法不添加任何新的状态, 但是运行完 ϵ 移除算法之后, 原来的状态不是每个都需要了, 有些状态可能变成不可达, 因此在算法运行完之后, 还需要使用裁剪 (Trimming) 算法^[6] 把这些无效的状态和边去除, 如图 15-18、图 15-19、图 15-20 所示。

图 15-18 ϵ 移除运行之前^[9]图 15-19 ϵ 移除运行完之后。其中状态 2,3 不可达，需要移除^[9]图 15-20 ϵ 移除运行完成，裁剪后的结果^[9]

5. 因子分解

因子分解 (Factorization) 操作把一连串的链式转移替换为单个的转移，输入输出符号只需要拼接起来即可，权重使用乘法操作得到新的权重，这样可以大大减少 WFST 图的大

小。如图 15-21 和图 15-22 所示分别是因子分解之前和分解之后的 WFST 示意图。

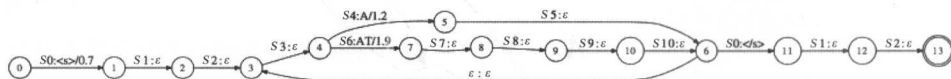


图 15-21 因子分解之前的 WFST^[1]

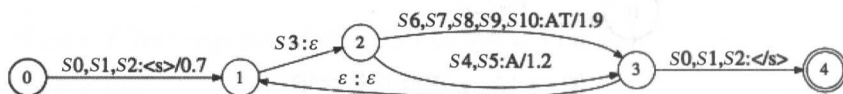


图 15-22 因子分解之后的 WFST^[1]

15.4 基于 WFST 的语音识别系统

在大规模连续语音识别中，WFST 可以将各种资源整合为统一的整体，如 HMM 声学模型、三音素转换、发音字典、语言模型。WFST 可以使用前面提到的各种优化算法，得到等价的、更加高效的 WFST。由于消除了一些冗余，而且能提前预知一些量，所以大大加快了解码过程。

连续语音可以定义为在给定语音输入 O 的前提下，寻找最有可能的单词序列 \hat{W} ，使得后验概率 $p(\hat{W}|O)$ 最大。使用贝叶斯公式，而且由于单词可能存在多种发音，所以整个过程可以表示为如下数学形式：

$$\begin{aligned}\hat{W} &= \arg \max_{W \in \mathcal{W}} \sum_{V \in R(W)} p(O|V, W) p(V|W) p(W) \\ &= \arg \max_{W \in \mathcal{W}} \left\{ \sum_{V \in R(W)} p(O|V) p(V|W) p(W) \right\}\end{aligned}$$

其中， \mathcal{W} 为所有可能的单词序列。 $p(O|V, W)$ 为声学模型，通常我们假设在给定音素序列的条件下，观察序列 O 与单词序列 W 条件独立，即 $p(O|V, W) = p(O|V)$ ，声学模型可以使用传统的 GMM-HMM 来建模，也可以使用 HMM-DNN 深度模型来建模。 $p(V|W)$ 为在给定单词序列 W 下，其发音音素序列为 V 的概率，只有一种发音的单词，该概率值为 1.0；若有多种发音，则需要人工指定每种发音的概率。 $p(W)$ 为语言模型，一般通过统计得到，例如 n -gram，当然也可以使用 RNN 等神经网络来建模。

对于大部分单词，只有一种发音，而且即使有多种发音，通常也只有一种发音占主宰地位，因此可以把上式的对于所有 V 的求和近似为求最大值（Viterbi 近似）：

$$\hat{W} \approx \arg \max_{W \in \mathcal{W}} \left\{ \max_{V \in R(W)} p(O|V)p(V|W)p(W) \right\}$$

取负对数，问题从乘积变为求和、从求最大值变成求最小值：

$$\hat{W} \approx \arg \min_{W \in \mathcal{W}} \left\{ \min_{V \in R(W)} \left\{ -\log(p(O|V)) + -\log(p(V|W)) + -\log(p(W)) \right\} \right\}$$

下面我们试图使用 WFST 来解决以上求最优单词序列的问题。

（1）声学模型 WFST：输入为观察序列，输出为上下文相关三音素序列，转移权重等于上式中的 $-\log(p(O|V))$ ，该 WFST 一般记为 H 。

（2）发音字典 WFST：输入为单音素序列，输出为单词，转移权重等于上式中的 $-\log(p(V|W))$ ，该 WFST 一般记为 L 。

（3）语音模型 WFST：输入为 n -gram 的单词序列，输出也是单词序列，转移权重等于上式中的 $-\log(p(W))$ ，该 WFST 一般记为 G 。

上式中的“+”操作就是热带半环中的乘法 \otimes ，“min”操作就是热带半环中的加法 \oplus ，组合操作中的权重是采用 \otimes 操作的，所以上式可以表示为在完全组合的统一 WFST 上求最小权重的路径序列问题。

$$\begin{aligned} \hat{W} &\approx \arg \min_{W \in \mathcal{W}} \left\{ \min_{V \in R(W)} \left\{ -\log(p(O|V)) + -\log(p(V|W)) + -\log(p(W)) \right\} \right\} \\ &= \arg \min_{W \in \mathcal{W}} \left\{ \min_{V \in R(W)} \left\{ w_H(O \rightarrow V) \otimes w_L(V \rightarrow W) \otimes w_G(W \rightarrow W) \right\} \right\} \end{aligned}$$

在三音素上下文相关模型中，还需要一个从三音素转到单音素的转换器，记为 C ，上式变为：

$$\begin{aligned} \hat{W} &\approx \arg \min_{W \in \mathcal{W}} \left\{ \min_{V \in R(W)} \left\{ w_H(O \rightarrow V) \otimes w_C(V \rightarrow V) \otimes w_L(V \rightarrow W) \otimes w_G(W \rightarrow W) \right\} \right\} \\ &= \arg \min_{W \in \mathcal{W}} \left\{ \min_{V \in R(W)} \left\{ w_N(O \rightarrow W) \right\} \right\} \end{aligned}$$

其中 N 为完全组合的统一 WFST：

$$N = H \circ C \circ L \circ G$$

完全组合的 WFST 形成了一个一体化的搜索网络，考虑跨词三音素（Cross-word Tri-phone）特别容易集成进去，这种情况在传统方法中是非常复杂的。而且完全组合的 WFST 还可以进一步优化得到等价的 WFST，能够极大地提高搜索效率，这在传统的解码方法中是很难实现的。

完全组合的 WFST 一旦建立，就基本不需要更新，对于任意的语音输入，解码器的工作就是专心在 WFST 上找到一个最优的路径，不像传统的方法还需要动态扩展搜索网络。除非 H, C, L, G 中的某个 WFST 要更新了，这时就需要重新做一次 WFST 完全组合。

WFST 还有一个优势，就是允许作为其他目的去使用解码器。例如，如果只需要从语音信号得到音素序列，那么组合 $H \circ C$ 就行；或者中间插入一个步骤时，如果该步骤能定义一个 WFST，那么就只需要重新做一次完全组合就行。在传统的方法中这些代码都需要重写。下面将介绍如何构建语音识别的 WFST 组件及其组合和优化，以及基于 WFST 的语音识别解码器是如何做搜索的。

15.4.1 声学模型 WFST

声学模型可以看成一种从语音信号转成上下文相关音素（三音素）单元的 WFST。如果只考虑 sst 音素序列，那么对应的就会有 $s(t), (s)s, (s)s(t)$ 上下文相关的三音素。如果使用三状态的 HMM 来建模的话，因为每个 HMM 天然就是一个自动机，因此只需要对这三个 HMM 做并操作（Union），就能得到如图 15-23 所示的 HMM 转换器。

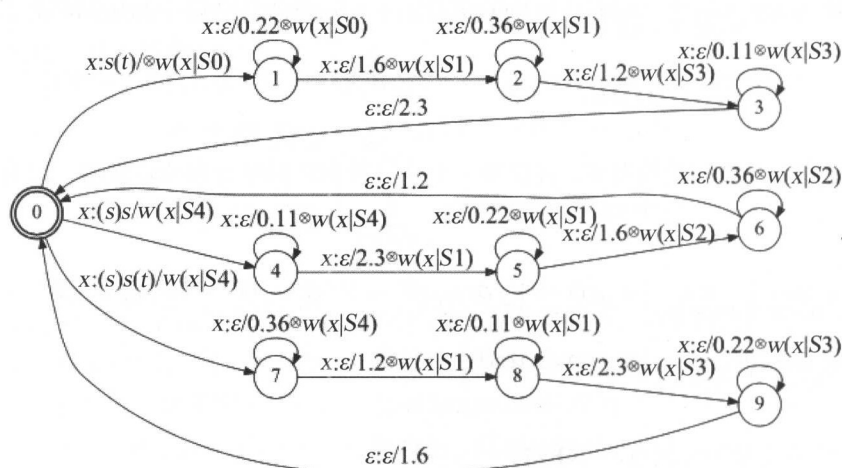


图 15-23 HMM 非标准的 WFST^[1]

(1) $(s)s(t)$ 转移表示在音素 s 左右两边的音素分别是 s, t ，如果左右两边没有括号，就表示没有相应方向的上下文。

(2) x 为元信息，代表所有特征空间的向量集合。这个元信息很重要，否则特征向量是连续不可数的，在 WFST 上是无法表达的。

(3) $x : s(t)/w(x|S0)$ 表示从状态 0 到 1 的输入是 x ，输出是 $s(t)$ ，权重为 $w(x|S0)$ ，其中 $S0$ 表示第 0 个共享状态，三个模型的中间状态是共享的，都是 $S1$ ，每个三音素的 HMM 的共享状态 ID 都是事先知道的。

(4) $w(x|Sk)$ 在热带半环和对数半环中表示 $-\log(b_k(x))$ 。

(5) 状态 1 的自旋 $x : \varepsilon/0.22 \otimes w(x|S0)$ ，其中 $0.22 = -\log 0.8$ 。

如图 15-23 所示的不是标准的 WFST 定义，因此不能直接使用 WFST 的框架。由于有连续的特征向量空间无法直接定义在 WFST 上，这里输入用元符号 x 来表示，但是 $w(x|Sk)$ 的信息是无法提前预知的，所以需要根据当前时刻的声学特征向量来实时计算。

可以把上面的非标准转换器分解为两个部分，一部分是 HMM 拓扑转换器，是标准的 WFST，预先就可以构建好，其接收的是共享状态序列 ID，输出的是三音素序列，转移权重为负对数状态转移概率；另一部分是声学模型匹配转换器，需要实时构建，其输入为观察序列元信息 x ，输出为共享状态，转移权重为负对数发射概率 $w(x|Sk)$ ，需要根据声学特征向量实时计算。后面多个 WFST 统一组合时，只使用 HMM 拓扑 WFST，最后在解码过程中，声学匹配转换器才与统一组合的 WFST 进行实时组合操作，因为它的输出可以是任意的共享状态，所以组合不需要使用以上介绍的组合算法显式进行，只需要把权重做乘法 \otimes 就行。如图 15-24 和图 15-25 所示分别为 HMM 拓扑 WFST 和声学匹配转换器。

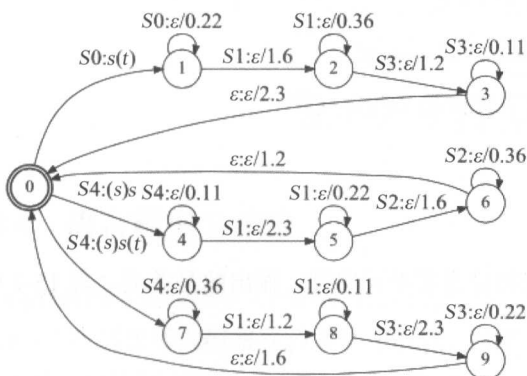


图 15-24 HMM 拓扑 WFST— $H^{[1]}$

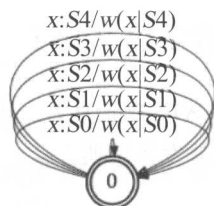


图 15-25 声学匹配转换器^[1]

15.4.2 三音素 WFST

基于三音素的上下文相关的模型被大多数语音识别系统所采用。从声学模型 WFST 输出的是三音素单元，因此需要一个转换器把三音素转换成正常的单音素，该转换器记为 C 。 C 的构建不难，三音素的上下文关系通过相连的两个状态及其转移构建，每个状态代表一个音素对，每个转移对应一个三音素，输出为三因素中间的字符。 C 的构建跟后面讲到的语言模型转换器 G 的构建有点类似，但是稍微有点不同，详细信息请参考 15.4.4 节。

对于三音素上下文依赖的 WFST，其中每个转移连接两个状态，源状态用 (left, center) 音素对表示，目的状态用 (center, right) 音素对表示，转移的输入为三音素，输出为上下文无关的单音素，权重为乘法单位元 $\bar{1}$ ，即没有权重，只是起到转换作用。如图 15-26 所示为只有两个音素单元 /t/ 和 /s/ 的转换器，其中状态 2 为 (s)s，状态 3 为 s(t)，它们之间的转移为 (s)s(t)，输出为单音素 s。

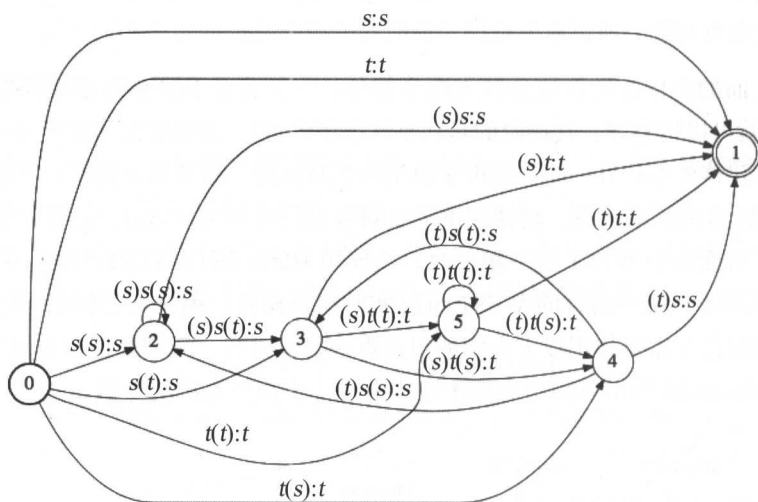


图 15-26 上下文依赖 WFST— $C^{[1]}$

15.4.3 发音字典 WFST

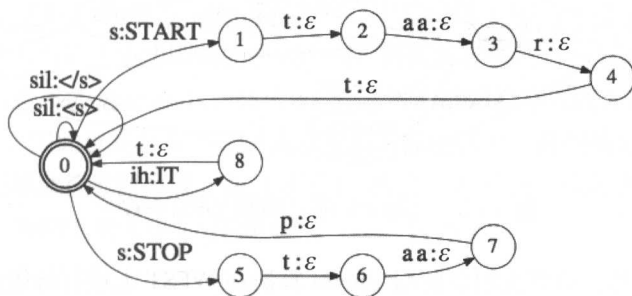
发音字典 WFST 记为 L ，其输入对应的是音素单元序列，输出的是单词。表 15-2 中显示的是英文部分发音字典。

表 15-2 英文部分发音字典

单词	发音
<s>	sil
</s>	sil
START	s t a a r t
STOP	s t a a p
IT	ih t

如图 15-27 所示对应的是表 15-2 中发音字典构建的 WFST，有几个特殊的单词，<s> 表示句子的开始，</s> 表示句子的结束，这两个符号在后面的语言模型中会用到。在这里还处理了在任意两个单词之间插入短暂停的情况，因为说话时，词与词之间确实会存在间隙，如果语言模型中有短暂停条目，那么需要在发音字典里也加上；如果语言模型中没有短暂停条目，发音字典则需要添加一种特殊的短暂停转移 sp:epsilon，这种转移没有输出，任意两个单词之间可以有 0 个或多个短暂停。

还有一种特殊情况是一个单词可能存在多种发音，如果单词 w 只有一种发音 v ，那么 $P(v|w)$ 的概率总是 1；如果单词有多种发音，那么这些发音概率值加起来要等于 1.0。在 L 热带半环 WFST 中，转移权重为 $-\log(P(v|w))$ 。

图 15-27 发音字典 WFST— $L^{[1]}$

15.4.4 语言模型 WFST

n -gram 模型是一种 $n-1$ 阶的马尔可夫模型，可以使用 WFST 来表示，一般记为 G 。理论上 $n-1$ 阶马尔可夫模型有 $|V|^{n-1}$ 个状态、 $|V|^n$ 个转移，其中 $|V|$ 为词汇量。每个转移涉及的源状态可以用 w_{i-n+1}^{i-1} ， $n-1$ 个字符标识，目的状态可以用 w_{i-n+2}^i ， $n-1$ 个字符标识，转移三元组为 $(w_{i-n+1}^{i-1}, w_i, -\log(p(w_i|w_{i-n+1}^{i-1})))$ 。

在大规模连续语音识别场景中,词汇量 $|V|$ 一般很大,而且要得到很好的识别效果,一般 $n \geq 3$,因此所需要的内存跟随 n 呈指数级增长,跟随 $|V|$ 呈幂增长。例如 $|V| = 50000, n = 3$,如果每个转移用4字节表示的话,那么理论上所需要的内存大约为450TB。还好,以上计算的是理论上所需要的内存规模,但是 n -gram一般都是基于常用语料直接统计词频得到的,其中大部分的词出现次数很少或未出现,所以统计分布会出现长尾问题。

这个问题可以使用回退平滑机制(Back-off Smoothing)来解决^[1],对于那些没有见过的 n -gram使用 m -gram概率值乘以回退系数来计算概率,其中 $m < n$, m -gram的存储量要远小于 n -gram,因此在大规模连续语音识别场景中能正常使用。如图15-28所示就是一个采用回退平滑机制的bigram的简单WFST,"STOP IT"在bigram中能找到,状态4转移到状态6的权重为 $-\log P(IT|STOP) = 0.69$,而"START IT"在bigram WFST状态2上没能找到相应的转移边,但是可以先通过 ϵ 输入转移到状态2,最后再转移到状态6。以上 ϵ 转移其实就是执行回退的过程,因此 $-\log P(IT|START) = -\log \alpha(START) + -\log P(IT) = 1.79$ 。如果当前状态下找到任何的状态转移,则说明出现了未见过的词,因此需要通过一个 ϵ 转移,利用 $m-1$ 个词的 m -gram状态;如果 m -gram还是未见过,则可以继续递归使用回退概率,直到 m 等于0为止。

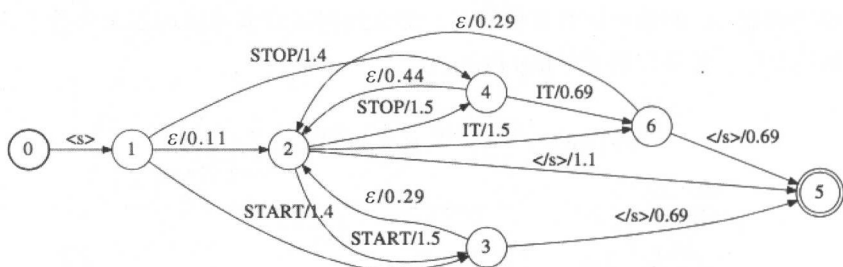


图 15-28 bigram 语言模型 WFST— $G^{[1]}$

有一点需要注意,后面我们要使用 Viterbi 算法在 WFST 上进行解码,最终是寻找一条概率最大的路径。由于回退机制的出现,计算同一个 n -gram 概率时可能会存在多条路径,所以需要遵循的一个原则是,不到迫不得已时不使用回退路径,因为 n -gram 路径和回退路径同时存在时,如果回退路径的概率比 n -gram 还要大的话,那么会造成解码错误。还好,在一般情况下, n -gram 的概率都会大于回退路径的概率。

15.4.5 WFST 组合和优化

以上我们介绍了语音识别系统中的 4 个 WFST, 即 HMM 声学模型 H 、三音素转换 C 、发音字典 L 和语言模型 G , 在最终用于解码前需要把这 4 个 WFST 组合成一个统一的 WFST, 因为使用统一的单个 WFST 可以通过优化算法得到等价的、更加高效的 WFST。而且统一的 WFST 一旦构建好了, 在 H, C, L, G 更新前是不需要重建的, 在线上解码前, 可以提前离线构建好。

一般的组合优化过程如下:

$$N = \text{fact}(\pi_{\epsilon}(\min(\det(\tilde{H} \circ \tilde{C} \circ \det(\tilde{L} \circ G))))))$$

步骤 1: $\det(\tilde{L} \circ G)$

如果把 n -gram 语言模型中的回退 ϵ 转移当成正常标签的话, 那么 G 就是确定化的, 所以不需要对 G 做确定化。由于不同的单词可能有相同的发音序列, 所以 L 不是确定的。

为了使得 L 是可确定化的, 一般的做法是在发音序列结尾处加入一个特殊的辅助符号。例如 night, knight 具有相同的发音序列 “n ay t”, 在这两个单词后面分别插入 #1 和 #2 特殊符号, 这样 night, knight 就有了不同的发音序列。同样, 对于有 M 个相同发音的情况, 只需要分别插入 #1, #2, ..., #M, 即 M 个辅助符号即可。但是这样 L 还不一定是确定化的, 因为有可能同一个音素序列对应于多个不同的单词序列。例如发音序列 “t ax n ay t”, 对应的单词序列可能是 tonight, 也可能是 to night, 插入 #1 后这两个单词的序列就分别变成 t ax n ay t #1 和 t ax #1 n ay t #1, 因此也需要在普通单词后面加入辅助符号, 得到确定化的 \tilde{L} (在最终确定化完成之后, 把辅助符号替换成 ϵ 就可以了)。然后与 G 组合, 最后再做确定化 (做确定化可以有效地减小 WFST 的大小), 得到 $\det(\tilde{L} \circ G)$, 记为 LG 。

步骤 2: $\tilde{C} \circ LG$

因为多个不同的三音素序列会对应到相同的单音素序列, 在执行 C 的组合操作前, 首先需要对 C 确定化, 使得 C 对输出是确定化的——相同的输出只能对应一个输入。之前我们只介绍了对输入做确定化, 对输出做确定化其实很简单, 只需要先把 C 输入输出反转过来, 得到 $\text{invert}(C)$, 然后做确定化, 得到 $\det(\text{invert}(C))$, 最后再反过来, 得到 $\text{invert}(\det(\text{invert}(C)))$, 记为 C' 。

在 C' 和 LG 做组合之前, 不要忘了在 C' 中也要插入辅助符号 #1, #2, ..., #M, 因为不这样做的话, 组合之后的 WFST 就不会有辅助符号了。如果在发音字典里面最多的具有相同发音的单词个数为 M , 那么只需要对 C' 中的每个状态加 M 个自旋转移 “#m:#m” 即可, 这

样就得到了含有辅助符号的确定化的三音素上下文 WFST，记为 \tilde{C} 。最后与步骤 1 的结果做组合，得到的结果记为 $CLG = \tilde{C} \circ LG$ 。因为 \tilde{C} 是对输出确定化的，因此 \tilde{C} 与 LG 做组合的大小不会比 LG 大多少，而且所得到的组合几乎是确定的，因此不需要再进行确定化操作。

步骤 3: $\tilde{H} \circ CLG$

跟 C 一样， H 也要加上辅助符号——只需要在初始状态上加上辅助符号的自旋转移即可。因为辅助符号是以音素为单位的，而不是以状态为单位的，加上辅助符号的 H 记为 \tilde{H} ，与 CLG 做组合操作和确定化操作，得到 $\det(\tilde{H} \circ CLG)$ ，记为 $HCLG$ 。

步骤 4: $\text{fact}(\pi_\epsilon(\min(HCLG)))$

对 $HCLG$ 先做最小化操作，得到等价的最少数量状态的 WFST，然后把之前加入的辅助符号替换成 ϵ ，最后使用因子分解操作来得到体积更小的 WFST。虽然 fact 操作减少了状态数量，但是实际上并没有减少太多的存储输入输出的空间，这样使得解码时代码变得复杂，因为一个转移可能包含多个状态，内部还得有一个循环处理。而且 fact 操作还有一个坏处就是压缩后时间信息丢失了，如果你想知道最优路径每个时间帧对应的状态，那么使用 fact 操作的 WFST 是没法得到的。在 Kaldi^[10] 里面没有使用 fact 操作。

15.4.6 组合和优化实验

如表 15-3 所示是在 CSJ (Corpus of Spontaneous Japanese) 语料上的 WFST 组合实验结果，该语音语料包含了 2000 个 10~30 分钟长度的课程语音。采用 GMM-HMM 模型，三音素声学模型有 5000 个共享的状态，发音字典覆盖了 10 万个左右的单词。表 15-3 呈现的是以上 WFST 组合和优化过程中的状态和转移的规模，从表中可以看出 G 是所有 WFST 中最大的，占用空间最多。由于含有相同发音的单词不是很多，所以 $\tilde{L} \circ G$ 组合操作不会比 G 或 L 大特别多。 \tilde{C} 是对输出确定化的，而且一个正常输入基本对应于一个正常输出，因此 \tilde{C} 与 LG 做组合的大小不会比 LG 大多少。由于 H 是采用三状态的，因此 $\tilde{H} \circ CLG$ 的状态数量几乎是 CLG 的 3 倍。确定化操作一般能使得 WFST 变小一点，最小化和因子分解操作使得 WFST 的图大小变小最多。

表 15-3 CSJ 数据集上的组合操作实验结果^[1]

WFST 组合和优化操作	# 状态	# 转移
H	6518	26634
C	1894	85185
L	501759	602566
G	220773	1177625

续表

WFST 组合和优化操作	# 状态	# 转移
$\tilde{L} \circ G$	1510293	2849833
$LG = \det(\tilde{L} \circ G)$	1184192	2229092
$CLG = \tilde{C} \circ LG$	1195569	2318779
$\tilde{H} \circ CLG$	3145430	4268640
$HCLG = \det(\tilde{H} \circ CLG)$	3034175	4102668
$\pi_\epsilon(\min(HCLG))$	2698861	3667087
$N = \text{fact}(\pi_\epsilon(\min(HCLG)))$	736221	1704447

15.4.7 WFST 解码

前面我们已经了解了如何把语音识别的流程整合到统一的 WFST 上, 下面继续介绍如何在统一的 WFST 上做语音解码, 即给定一段语音输入 $O = o_1, \dots, o_T$, 如何解码成一段对应的文字序列。

在前面章节中我们介绍了 Viterbi 解码, 使用 Viterbi 解码可以得到最优的状态序列, 前提是我们给定了文本。但是在语音识别中, 我们事先肯定不知道语音对应的文本, 那么使用基于动态规划的 Viterbi 解码是行不通的。对于所给定的一段语音, 将其解码成任何文本序列都是有可能的, 只是解码成不同的文本序列的概率大小不同而已。由于任意的时间 t 都可能处于任意单词 w 的任意音素 q 的任意状态 j , 因此, 我们也可以使用动态规划法写出类似的 Viterbi 解码递归式:

$$\alpha(t, w, q, j) = b_j(t) \max \left\{ \alpha(t-1, w, q, i) a_{ij}, \max_{w' \in \mathcal{W}} \{ p(w|w') \alpha(t-1, w', q_f, i_f) a_{i_f j} \} \right\}$$

其中, q_f 为单词 w' 的最后一个音素, i_f 为 q_f 的结束状态。要精确算出上面递归式, 时间和空间复杂度特别高, 根本没法在大规模词汇的语音识别场景中使用。因此, 我们需要使用 Beam Search 来做贪心近似, 即在上式的递归关系中, 不考虑词汇表中的所有词汇, 只考虑 n -gram 概率大的词汇, 在每个时间点及时去除那些概率特别小的路径, 这样整个搜索空间大小就降下来, 解码变得可用了。我们将这种结合 Viterbi 和 Beam Search 的宽度优先的解码算法称为 Time-Synchronous Viterbi Beam Search 算法, 其中 Viterbi 提供了状态间的最优子结构, Beam Search 提供了贪心剪枝。具体的算法细节请参考文献 [1]。

参考文献

- [1] Hori, Takaaki, and Atsushi Nakamura. Speech recognition algorithms using weighted finite-state transducers. *Synthesis Lectures on Speech and Audio Processing* 9.1 (2013): 1-162.
- [2] Mohri, Mehryar, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language* 16.1 (2002): 69-88.
- [3] Mohri, Mehryar. Finite-state transducers in language and speech processing. *Computational linguistics* 23.2 (1997): 269-311.
- [4] Allauzen, Cyril, et al. OpenFst: A general and efficient weighted finite-state transducer library. *Implementation and Application of Automata* (2007): 11-23.
- [5] Mohri, Mehryar. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics* 7.3 (2002): 321-350.
- [6] Mohri, Mehryar. Weighted automata algorithms. *Handbook of weighted automata* (2009): 213-254.
- [7] Mohri, Mehryar, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. *Springer Handbook of Speech Processing*. Springer Berlin Heidelberg, 2008. 559-584.
- [8] Mohri, Mehryar. Minimization algorithms for sequential transducers. *Theoretical Computer Science* 234.1-2 (2000): 177-201.
- [9] <http://www.gavo.t.u-tokyo.ac.jp/~novakj/WFST-algorithms.pdf>.
- [10] Povey, Daniel, et al. The Kaldi speech recognition toolkit. *IEEE 2011 workshop on automatic speech recognition and understanding*. No. EPFL-CONF-192584. IEEE Signal Processing Society, 2011.

基于 GMM-HMM 的传统语音识别模型在深度语音识别技术出现以前，至少统治了 20 年的时间。但是随着深度学习浪潮的席卷，目前基本上所有的 state-of-the-art 的语音识别模型都是基于深度学习的模型，或者是深度学习和传统 HMM 模型相结合的双向模型。2012 年，俞栋、邓力、Hinton 等人第一次将深度前向神经网络应用到语音识别中，相比于传统的 GMM-HMM 模型识别效果有很大的提升，后来陆续有学者将深度 CNN, RNN, LSTM 模型成功地应用到语音识别中。下面我们将介绍一些经典的与深度语音识别相关的工作。

16.1 CD-DNN-HMM

基于人工神经网络 (ANN) 和 HMM 的语音识别模型早在 20 世纪 80 年代末就出现了，但是效果相比于 GMM-HMM 提升较小，它没有流行起来可能有如下几个原因。

- (1) 受限于计算资源，那时候的人工神经网络很少有超过 2 层的。
- (2) 由于梯度发散问题，BP 算法在隐层超过 2 层的时候很难训练。
- (3) 在 GMM-HMM 中基于上下文三音素模型开发的很多技术，很难应用起来。

前面章节我们介绍了，基于上下文相关的语音识别模型相比于上下文无关的语音识别模型在效果上会有很大的提升，早期的基于上下文相关的 ANN-HMM 模型使用了如下两种后验概率建模策略：

$$p(x_t, c_i | o_t) = p(c_i | x_t, o_t) p(x_t | o_t)$$

$$p(x_t, c_i | o_t) = p(x_t | c_i, o_t) p(c_i | o_t)$$

其中 o_t 为观察变量, c_i 为第 i 个聚类后的上下文相关分类, x_t 为上下文无关的状态。

如果直接对 x_t 和 c_i 的笛卡儿积 $p(x_t, c_i | o_t)$ 建模的话, 分类数量为 JK , 其中 J, K 分别为聚类后的上下文相关的分类数量和上下文无关的状态数量, 建模参数规模会特别大, 很容易造成过拟合, 而且很多上下文在训练数据中是覆盖不到的。而上面的后验概率建模策略采用概率的链式法则分解成 $p(x_t | o_t)p(c_i | x_t, o_t)$ 两部分进行层次建模, 这样总的分类数量就降到 $J + K$, 参数规模变小很多。虽然这种后验概率建模策略在一些任务中比 GMM-HMM 效果要好, 但是改善并不大。

如图 16-1 所示是 2012 年俞栋、邓力、Hinton 等人提出的 CD-DNN-HMM 双向模型, 跟早期的 ANN-HMM 设计有如下几点不同。

- (1) 对聚类后的状态 (也称为共享状态) 直接建模, 使用统一的深度神经网络来建模。
- (2) 把传统的浅层神经网络模型 ANN, 换成了深层神经网络模型 DNN。
- (3) DNN 的输入特征不仅使用了当前帧的信息, 而且还使用了前后相邻帧的信息 (一般窗口大小为 9~13 帧)。

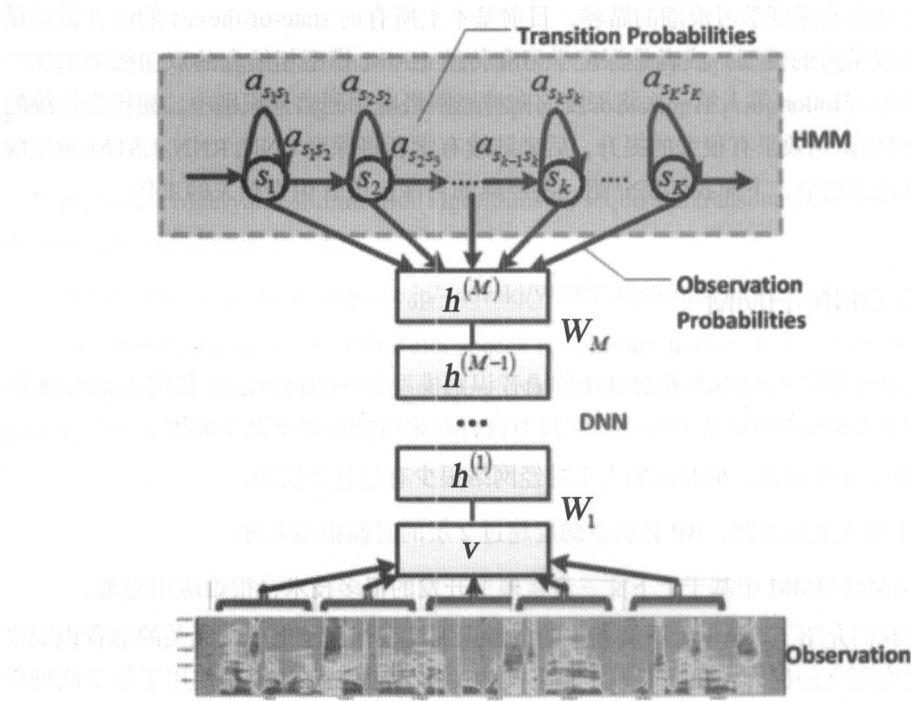


图 16-1 DNN-HMM 模型, 其中 HMM 为语音时序信号建模, DNN 为后验概率建模^[1]

在 GMM-HMM 模型中, 使用 GMM 为似然 $p(o_t|x_t)$ 建模, 但是直接使用 DNN 来为 $p(o_t|x_t)$ 建模是比较困难的, 利用贝叶斯公式:

$$p(o_t|x_t) = \frac{p(x_t|o_t)p(o_t)}{p(x_t)}$$

其中, $p(x_t|o_t)$ 为给定的观察值 o_t 、状态 x_t 的后验概率, 可以使用统一的 DNN 建模。 $p(x_t)$ 是共享状态的边际概率, 可以使用 GMM-HMM 对训练数据做强制对齐, 然后直接统计得到。虽然除以先验可能在一定条件下不能提升识别精度, 但是在一些含有长静音 (Silence) 的训练语料中去除偏差 (Bias) 还是非常有用的。也就是说, 如果含有大量的静音, 后验概率算出来的静音的概率天然就会偏大, 但是除以先验之后, 就变成无偏的了。 $p(o_t)$ 为观察值的概率, 它与最终解码出来的词序列无关, 在解码时可以忽略掉, 那么以上的似然就变成了缩放形式的似然:

$$\bar{p}(o_t|x_t) = \frac{p(x_t|o_t)}{p(x_t)}$$

使用缩放形式的似然 $\bar{p}(o_t|x_t)$ 并不会影响解码结果:

$$\begin{aligned} \tilde{W} &= \arg \max_{W \in \mathcal{W}} p(W|O) \\ &= \arg \max_{W \in \mathcal{W}} \frac{p(O|W)p(W)}{p(O)} \\ &= \arg \max_{W \in \mathcal{W}} p(O|W)p(W) \\ &= \arg \max_{W \in \mathcal{W}} \left\{ \sum_X p(x_1) \left[\prod_{t=2}^T p(x_t|x_{t-1}) \right] \prod_{t=1}^T p(o_t|x_t) \cdot p(W) \right\} \\ &\approx \arg \max_{W \in \mathcal{W}} \left\{ \max_X \left\{ p(x_1) \left[\prod_{t=2}^T p(x_t|x_{t-1}) \right] \prod_{t=1}^T p(o_t|x_t) \right\} \cdot p(W) \right\} \\ &= \arg \max_{W \in \mathcal{W}} \left\{ \max_X \left\{ \log p(x_1) + \sum_{t=2}^T \log p(x_t|x_{t-1}) + \sum_{t=1}^T \log p(o_t|x_t) \right\} + \log p(W) \right\} \\ &= \arg \max_{W \in \mathcal{W}} \left\{ \max_X \left\{ \log p(x_1) + \sum_{t=2}^T \log p(x_t|x_{t-1}) + \sum_{t=1}^T \log \bar{p}(o_t|x_t) \right\} + \log p(W) \right\} \end{aligned}$$

CD-DNN-HMM 使用 DNN 为聚类后的状态建模, 不仅解决了 DNN 不能为似然直接建模的问题, 而且还带来了两个其他的好处。

(1) CD-DNN-HMM 其实就是替换了传统 CD-GMM-HMM 中的发射概率建模, 那么

HMM 中的转移概率是可以直接复制过来使用的,一般不需要更新转移概率,而且基于 GMM-HMM 的状态聚类也是可以直接拿过来使用的,直接基于 DNN-HMM 做聚类会非常复杂。因此只需要对传统的 CD-GMM-HMM 模型做少量的修改,就可以很快地实现 CD-DNN-HMM 模型。

(2) 从上式可以看出,后验概率建模越准,给定单词序列的似然会越大,一般来说,语音识别精度就会越高,因此任何对 CD-GMM-HMM 模型的改善都能适用于 CD-DNN-HMM 模型,例如跨词的三音素模型、判别式训练等。

使用 DNN 为后验概率 $p(x_t|o_t)$ 建模,给定观察值 o_t 、预测状态 x_t 的后验概率,这是一个典型的多分类问题,多分类是监督学习问题,那么状态的标注数据从何而来?人工标注不但成本高,而且还不一定标得准确,显然是行不通的。可以使用一个训练好的 GMM-HMM 模型,采用 Viterbi 算法来做强制对齐,这样可以得到每个语音帧所属的状态,而标注数据的好坏直接影响 DNN 的性能,因此训练一个质量好的 GMM-HMM 模型是非常有必要的。

解决了训练数据的标注问题,接下来需要解决深度神经网络层数很深、BP 难以训练的问题。参考文献 [1] 中使用了 DBN (Deep Belief Network) 自底向上来做预训练,可以为深度神经网络进行很好的初始化,最后使用 BP 算法对 DNN 参数调优。

以上描述的训练算法称为嵌入式 Viterbi 算法,该算法的流程大致如下:

- (1) 训练一个 CD-GMM-HMM 模型,记为 HMM0。
- (2) 对所有的训练数据使用 HMM0 模型做 Viterbi 强制对齐。
- (3) 生成 DNN 训练数据的特征和标签。
- (4) 使用 DBN 预训练。
- (5) 统计聚类后状态的先验概率 $p(x)$ 。
- (6) 使用 BP 继续训练 DNN。

(7) 根据需要可以使用目前训练好的 DNN-HMM 模型,对 HMM0 中的转移概率做重新估计。

(8) 根据需要可以使用目前训练好的 DNN-HMM 模型,对训练数据进行重新对齐。

(9) 返回 CD-DNN-HMM。

CD-DNN-HMM 相比于 CD-GMM-HMM 在各种数据集上有很大的提升,句子准确率绝对值有 5.8%~9.2% 的提高,相对值有 16%~23.2% 的提高,根据实验结果得出了很多具有指导意义的结论^{[1][2][13]}。

(1) CD-DNN-HMM 模型的能力很强, 能为更多的训练数据建模, 共享状态的数量可以从原来的几百增加到几千的规模, 能够获得更加细致的标注。

(2) 三音素建模非常有效, 可以让模型从更加细致的标注数据中获得收益, 通过状态聚类又可以缓解过拟合, 在 SWB 任务中相比于单音素可以降低 50% 的相对错误率, 可见三音素建模收益非常明显。

(3) DNN 深度越深越好, 在必应移动搜索数据集上错误率从单层的 31.9% 降到了五层的 29.7%。

(4) 无论是浅层还是深层 DNN, 使用相邻帧信息作为特征输入, 准确率都有很大的提升, 在 SWB 数据集上深层 DNN 有 24% 的相对提升。

(5) 预训练在 DNN 层数小于 5 的时候确实有很大的收益, 但是随着层数的增加, 提升变小了。目前主流的深度学习都不需要使用预训练了。

(6) 训练数据的标注质量越好, CD-DNN-HMM 的训练效果就越好, 使用判别式训练的 CD-GMM-HMM 比使用最大似然的效果要好, 可以使用训练好的 CD-DNN-HMM 重新对训练数据做切分, 效果能够得到进一步提升。

(7) 重新调整 CD-DNN-HMM 的转移概率的作用并不明显。

CD-DNN-HMM 模型只是一个开始, 随后出现了大量的改进, 例如在深度 DNN 中加入判别式训练^[3], 把 DNN 替换成 RNN、双向 RNN、CNN 等神经网络模型, 后面我们会挑一些经典的工作进行详细介绍。

16.2 TDNN

RNN 是语音时序动态建模的最常用模型, 使用动态变化的上下文窗口, 可以捕获声学长依赖关系, 使用 RNN 的声学模型能达到目前最好的语音识别效果。但是由于 RNN 隐节点直接有顺序依赖关系, 非常不利于并行处理, 不能充分利用 GPU 这种单指令多数据的处理方式带来的并行化处理, 特别是在使用 LSTM 这样复杂记忆单元的时候, 因此 RNN 相比于简单的前向神经网络, 运算速度会非常慢。

早在 1989 年 Waibel 等人^[10]就提出了时延神经网络 (TDNN), 这种网络也可以高效地为长依赖时序关系建模, 而且在音素识别上取得了不错的效果。如图 16-2 所示, 在目前看来, 其实 TDNN 就是一种特殊的卷积神经网络, 第一层是一种卷积核大小为 16×3 (16 为归一化的梅尔系数)、跨度为 1 的卷积网络, 第二层是一种卷积核大小为 8×5 、跨度为 1 的卷积网络, 第三层是一种卷积核大小为 1×9 、跨度为 1 的卷积网络。

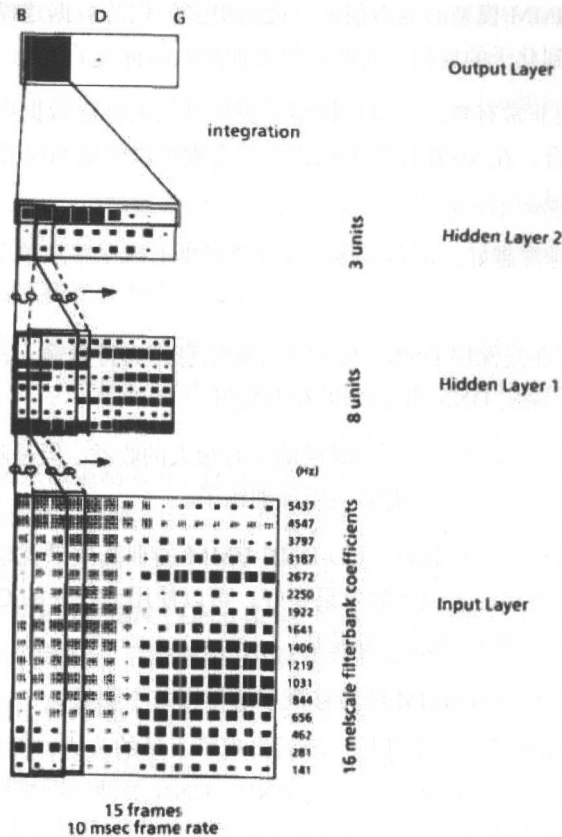
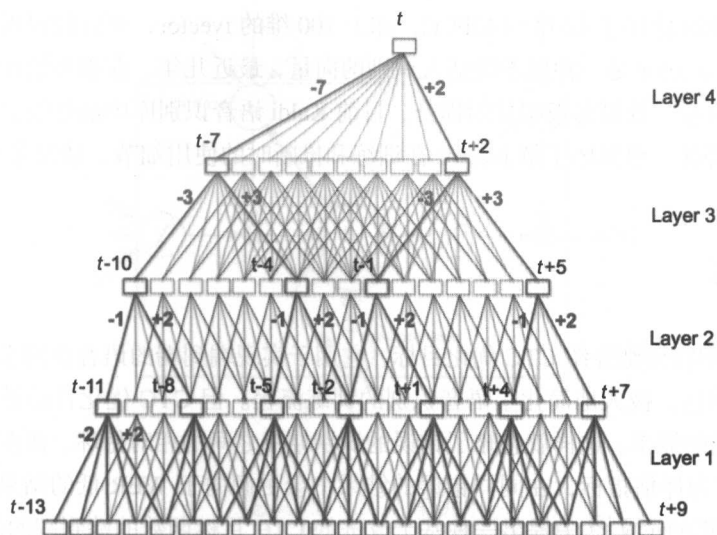


图 16-2 经典的 TDNN^[10]

虽然经典的 TDNN 是一种前向网络，但是计算还是很耗费机器资源的。Peddinti、Povey 等人提出了基于次采样的改进 TDNN^[4]，使用次采样技术，在保持经典的 TDNN 全部上下文信息的前提下，在每一个隐层只会挑选少量的节点参与计算，这样可以节省大量的计算资源。

如图 16-3 所示，深灰色线表示基于次采样技术参与计算的节点及其连接关系，浅灰色线表示没有次采样的经典 TDNN 参与计算的节点及其连接关系，从图中可以看出，经典的 TDNN 在相邻的隐节点中存在着大量的重叠信息，即相邻的隐节点是相关的，次采样 TDNN 就是基于这种观察，前向传播时过滤掉了一些节点，其中浅灰色线与深灰色线数量的比，就是传统的经典 TDNN 和次采样 TDNN 的计算量之比，浅灰色框与深灰色框数量之比就是占用存储资源量之比。可以明显地看出，基于次采样的 TDNN 的前向计算量、后向传播计算量、节点数量相比于经典的 TDNN 要少很多。

图 16-3 次采样 TDNN^[4]

次采样 TDNN 使用非对称的上下文输入，使用更多的左边上下文、相对较少的右边上下文，这样可以得到更好的识别精度，如表 16-1 所示为 TDNN 各层前后使用不同大小的上下文信息的实验效果对比情况。最终 4 层的 TDNN，每帧前后各使用 13 帧和 9 帧的上下文信息效果是最好的，如果使用再多的上下文信息，只会使帧的识别精度得到提高，但是对最终的 WER 的降低是不利的。跟传统的 TDNN 还有一点最大的不同，就是次采样使用的是 p -norm 非线性单元，而最近的实验指出，使用 ReLU 非线性单元的效果会更好。

表 16-1 TDNN 实验效果对比^[4]

Model	Network Context	Layerwise Context					WER	
		1	2	3	4	5	Total	SWB
DNN-A	$[-7, 7]$	$[-7, 7]$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	22.1	15.5
DNN-A ₂	$[-7, 7]$	$[-7, 7]$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	21.6	15.1
DNN-B	$[-13, 9]$	$[-13, 9]$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	22.3	15.7
DNN-C	$[-16, 9]$	$[-16, 9]$	$\{0\}$	$\{0\}$	$\{0\}$	$\{0\}$	22.3	15.7
TDNN-A	$[-7, 7]$	$[-2, 2]$	$\{-2, 2\}$	$\{-3, 4\}$	$\{0\}$	$\{0\}$	21.2	14.6
TDNN-B	$[-9, 7]$	$[-2, 2]$	$\{-2, 2\}$	$\{-5, 3\}$	$\{0\}$	$\{0\}$	21.2	14.5
TDNN-C	$[-11, 7]$	$[-2, 2]$	$\{-1, 1\}$	$\{-2, 2\}$	$\{-6, 2\}$	$\{0\}$	20.9	14.2
TDNN-D	$[-13, 9]$	$[-2, 2]$	$\{-1, 2\}$	$\{-3, 4\}$	$\{-7, 2\}$	$\{0\}$	20.8	14.0
TDNN-E	$[-16, 9]$	$[-2, 2]$	$\{-2, 2\}$	$\{-5, 3\}$	$\{-7, 2\}$	$\{0\}$	20.9	14.2

次采样 TDNN 使用了 40 维的 MFCC，加上 100 维的 ivector，可以使得识别效果有进一步的明显的提升。ivector 是一种用于说话人识别的向量，最近几年，在很多语音识别中都加入了 ivector 特征向量，效果有很明显的提升，目前 Kaldi 语音识别库中最好的语音识别模型就包含了 ivector 特征。想具体了解 ivector 在训练和推断时的使用细节，请参考文献 [11][12]。

16.3 CTC

在第 17 章中将详细介绍 CTC 优化目标，它用于实现端到端的语音识别系统，无需语音和文字的对齐信息，极大地简化了语音识别的训练流程。但 CTC 优化目标是最大化整个识别结果完全正确的概率，这样就忽略了错误的识别结果之间的相对好坏，而在语音识别中通常使用 WER 作为评估指标，在同样错误的结果中会更倾向于 WER 低的结果。于是，Alex Graves 在 2014 年的论文^[5]中提出一种基于改进的 CTC 目标函数的端到端的语音识别系统，其优化目标更接近于系统最终的评估指标。

在这个系统中，网络结构采用了双向的 RNN（如图 16-4 所示），从而可以利用双向的信息，RNN 内部使用 LSTM 单元（如图 16-5 所示）来更好地建模较长的依赖。这里使用了较深的网络结构，通过将多个 RNN 层堆叠到一起，有助于学习更高层的特征表示，如图 16-6 所示是一个单向深层 RNN，也可以将多个双向 RNN 层进行堆叠得到深层双向 RNN。

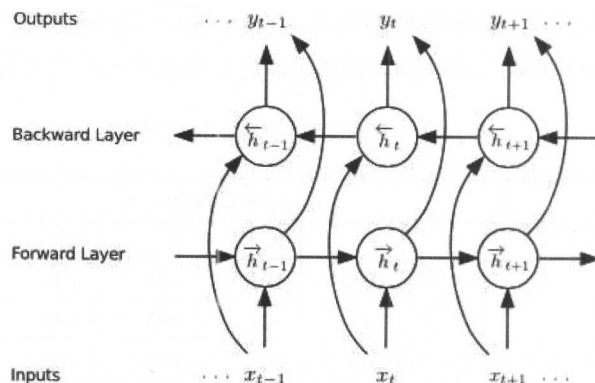
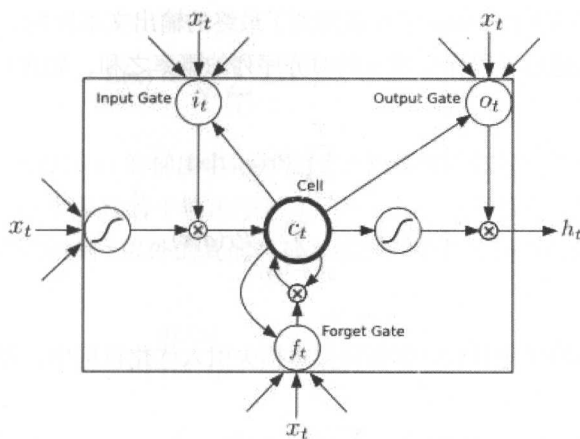
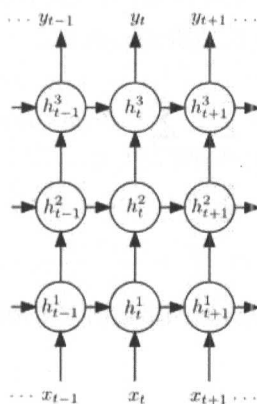


图 16-4 双向 RNN^[5]

图 16-5 LSTM 单元^[5]图 16-6 深层 RNN 结构^[5]

为了更好地理解改进的优化目标，我们简单回顾一下 CTC。给定一个长度为 T 的输入序列 x ，网络对每个时刻 t 预估一个向量 y_t ，用 softmax 进行归一化后得到 t 时刻生成各个标注（CTC 的建模单元，可以是字母或者音素）或 blank 的概率，如式 (16.1) 所示，其中 y_t^k 是向量 y_t 的第 k 个元素。

$$Pr(k, t) = \frac{\exp(y_t^k)}{\sum_{k'} \exp(y_t^{k'})} \quad (16.1)$$

CTC 的对齐序列 a 是一个长度为 T 的序列，序列中的每个元素都是 blank 或标注的下标，该序列的概率如式 (16.2) 所示。接下来对这个对齐序列进行 β 操作，就是先将相邻的重

复字符合并为一个，然后去掉 blank 字符就得到了最终的输出文本序列，于是模型生成文本序列 y 的概率为所有能通过 β 操作生成 y 的对齐序列的概率之和，如式 (16.3) 所示。

$$Pr(a|x) = \prod_{t=1}^T Pr(a_t, t|x) \quad (16.2)$$

$$Pr(y|x) = \sum_{a \in \beta^{-1}(y)} Pr(a|x) \quad (16.3)$$

下面我们来看改进的优化目标。作者将文本损失引入优化目标中，得到期望文本损失函数 $L(x)$ ，定义如下：

$$L(x) = \sum_y Pr(y|x) L(x, y) \quad (16.4)$$

其中， $Pr(y|x)$ 表示给定输入序列 x ，预测输出文本序列 y 的概率分布，它由 CTC 得到； $L(x, y)$ 是文本损失函数，如 WER。将式 (16.3) 代入式 (16.4) 可以得到如下结果，这个损失函数对不同的对齐序列 a 给予了不同的惩罚 $L(x, \beta(a))$ 。

$$\begin{aligned} L(x) &= \sum_y \sum_{a \in \beta^{-1}(y)} Pr(a|x) L(x, y) \\ &= \sum_a Pr(a|x) L(x, \beta(a)) \end{aligned} \quad (16.5)$$

为了训练模型，我们需要计算 $L(x)$ 对网络输出 y_t^k 的梯度，为此先计算 $L(x)$ 对 $Pr(k, t|x)$ 的梯度：

$$\begin{aligned} \frac{\partial L(x)}{\partial Pr(k, t|x)} &= \sum_a \frac{\partial Pr(a|x)}{\partial Pr(k, t|x)} L(x, \beta(a)) \\ &= \sum_{a: a_t=k} \frac{Pr(a|x)}{Pr(k, t|x)} L(x, \beta(a)) \\ &= \sum_{a: a_t=k} Pr(a|x, a_t=k) L(x, \beta(a)) \end{aligned} \quad (16.6)$$

由于 $L(x)$ 的计算要遍历所有可能的输出序列，计算量巨大，于是使用蒙特卡罗采样来近似计算 $L(x)$ 及其梯度。根据式 (16.5) 需要从分布 $Pr(a|x)$ 中采样若干序列 a ，根据式 (16.2) 该采样可以转换成每个时刻独立地从分布 $Pr(k, t|x)$ 中采样一个输出 a_t^i ，再拼接起来就可以得到一条样本 a^i ，如此采样 N 条样本用以计算近似的损失如下：

$$L(x) \approx \frac{1}{N} \sum_i^N L(x, \beta(a^i)), a^i \sim Pr(a|x)$$

同样, $L(x)$ 对 $Pr(k, t|x)$ 的梯度也可以使用蒙特卡罗采样进行近似, 根据式 (16.6) 需要从分布 $Pr(a|x, a_t = k)$ 中采样若干输出序列 a , 由于每个时刻的输出概率是独立的, 因此可以固定 $a_t = k$, 采样除时刻 t 以外的其他时刻的输出, 于是得到近似的梯度如下:

$$\frac{\partial L(x)}{\partial Pr(k, t|x)} \approx \frac{1}{N} \sum_{i=1}^N L(x, \beta(a^{i,t,k})) \quad (16.7)$$

上式中 $a^{i,t,k}$ 表示一个序列 $a^i, a^i \sim Pr(a|x)$ 并且该序列 a^i 的时刻 t 的输出为 k , 即 $a_t^i = k$ 。

进而, 我们可以计算 $L(x)$ 对网络输出 y_t^k 的梯度如式 (16.8) 所示, 其中 $\frac{\partial Pr(k', t|x)}{\partial y_k^k}$ 是 Softmax 函数的求导, $\delta_{k'k}$ 是指示函数, 当 $k' = k$ 时取值为 1, 否则为 0。

$$\begin{aligned} \frac{\partial L(x)}{\partial y_t^k} &= \sum_{k'=1}^K \frac{\partial L(x)}{\partial Pr(k', t|x)} \frac{\partial Pr(k', t|x)}{\partial y_k^k} \\ &= \sum_{k'=1}^K \frac{\partial L(x)}{\partial Pr(k', t|x)} (\delta_{k'k} - Pr(k', t|x)) Pr(k, t|x) \\ &= Pr(k, t|x) \left(\frac{\partial L(x)}{\partial Pr(k, t|x)} - \sum_{k'=1}^K Pr(k', t|x) \frac{\partial L(x)}{\partial Pr(k', t|x)} \right) \end{aligned} \quad (16.8)$$

将式 (16.7) 代入式 (16.8) 可以得到:

$$\frac{\partial L(x)}{\partial y_t^k} \approx \frac{Pr(k, t|x)}{N} \sum_{i=1}^N (L(x, \beta(a^{i,t,k})) - Z(a^i, t)) \quad (16.9)$$

其中 $Z(a^i, t) = \sum_{k'=1}^K Pr(k', t|x) L(x, \beta(a^{i,t,k'}))$ 。

从式 (16.9) 可以看出, $L(x)$ 对 y_t^k 的梯度正比于 $a_t^i = k$ 的文本损失和从分布 $Pr(k', t|x)$ 中采样 a_t^i 的期望文本损失这两者的差。这说明当改变输出序列 (对齐序列) 中一个时刻的输出可以改变损失时, 网络才会有相应的梯度。例如, 使用 WER 作为文本损失, 采样的输出序列经过 β 操作得到文本 “WTRD ERROR RATE”, 这时网络的梯度会倾向于将第二个输出 “T” 改成 “O”, 同时不改变另两个正确的单词。

如果网络采用随机初始化, 那么对模型的输出进行采样产生的几乎都是错误文本, 只改

变其中一个时刻的输出对损失的改变微乎其微，这时网络中的梯度值都非常小，训练很慢。所以，推荐先用 CTC 作为目标函数预训练一个网络使得模型的预估结果接近于真实，再使用这里提出的期望文本损失重新训练来进一步优化模型。

16.4 EESSEN

第 15 章介绍过在传统语音识别中可以利用 WFST 进行解码，这种方法预先对解码图进行了优化，解码更加高效。Y Miao 在参考文献 [6] 中首次将 WFST 用于 CTC 解码（该语音识别系统称为 EESSEN），使得 CTC 解码也可以有效地把字典和语言模型融合进来。类比于传统语音识别中的搜索图 HCLG，这里将 CTC 的 β 操作、字典及语言模型分别用一个 WFST 表示，然后将这三个 WFST 融合成一个搜索图 TLG 用于解码。下面具体介绍 CTC 解码所需要的各个 WFST。

1. 语言模型 (Grammar)

语言模型 WFST（称为 G）表示所有可能的单词序列并给出相应的序列在语言模型中的概率。该 WFST 的输入输出符号都是单词，边上的权重是语言模型中的概率，这和传统语音识别解码中的语言模型 WFST 相同。如图 16-7 所示是一个简化的 2-gram 语言模型 WFST 示意图，它可以接受两个句子：“how are you” 和 “how is it”，边上的权重表示语言模型中给定前一个词得到当前词的概率。

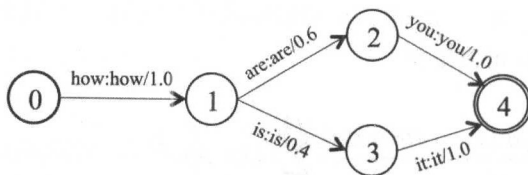


图 16-7 简化的语言模型 WFST 示意图^[6]

2. 字典 (Lexicon)

字典 WFST（称为 L）表示从字典基本单元到单词的映射关系。由于 CTC 既可以建模音素，也可以建模字符，因此对应地有两种字典 WFST：当 CTC 的输出是音素时，字典 WFST 和传统语音识别中的相同，表示从音素到单词的映射；当 CTC 的输出是字母时，该 WFST 中包含的是单词的拼写，这种字母字典图可以很方便地进行扩展，使其接受没见过的词（OOV, Out-Of-Vocabulary），而且在 CTC 的建模单元是字母时，通常需要给标注加入空格字符用于

间隔不同的单词，因此在音素 WFST 中也允许单词的开头和结尾有可选的空格。如图 16-8 和图 16-9 所示分别是音素发音字典 WFST 和字母拼写字典 WFST 的示意图。

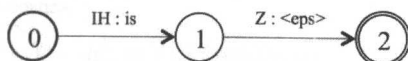


图 16-8 单词“is”的音素发音字典 WFST 表示，接受输入音素“IH Z”，得到输出单词“is”，<eps> 表示空^[6]

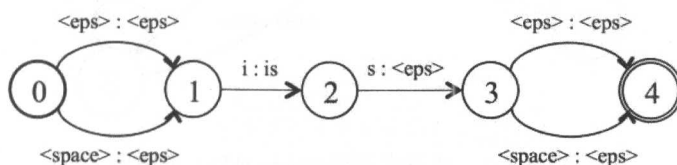


图 16-9 单词“is”的字母拼写字典 WFST 表示，这里允许在单词的开头和结尾可选地插入空格 <space>^[6]

3. Token

Token WFST（称为 T）表示 CTC 的 β 操作，它将 CTC 帧级别的输出序列映射成 β 操作后的字典基本单元（音素或字符）序列。例如对于 5 帧的语音输入，模型产生了三种可能的输出序列，即“AAAAA”“ $\emptyset\emptyset AA\emptyset$ ”和“ $\emptyset AAA\emptyset$ ”（其中 \emptyset 表示 blank），经过 Token WFST 这三个序列会得到输出“A”。如图 16-10 所示是一个基于字符的 Token WFST 示意图，节点 0 有若干条自旋边，接受空输入输出消歧符号 #1, #2, ..., #N，这些边是为了能和其他图做复合操作引入的；节点 2 和 3 分别是 CTC 标注的 <space>, A 符号，图中省略了 CTC 的其他标注单元，如 B, C, ..., Z。可以看到节点 3 将若干个连续的空格或 A 字符合并成一个，并将输入的 <blank> 符号转换为 <eps>，即输出去掉了 <blank>，这和 β 操作的结果一致。

4. 搜索图

得到上面三个 WFST 后，将它们融合到一起便可以得到最终的搜索图。首先将字典 WFST 和语言模型 WFST 复合（Composite）在一起，并对其进行确定化和最小化操作来压缩搜索空间，有助于提高解码速度。然后将所生成的 LG WFST 与 Token WFST 复合在一起，生成最终的搜索图。整个操作可以表示成下面这个式子：

$$S = T \circ \min(\det(L \circ G))$$

其中 \circ 、 \det 和 \min 分别表示复合、确定化和最小化操作。最终生成的搜索图 S 可以将 CTC 帧级别的输出序列转化成一个单词序列，也就是我们需要的解码结果。

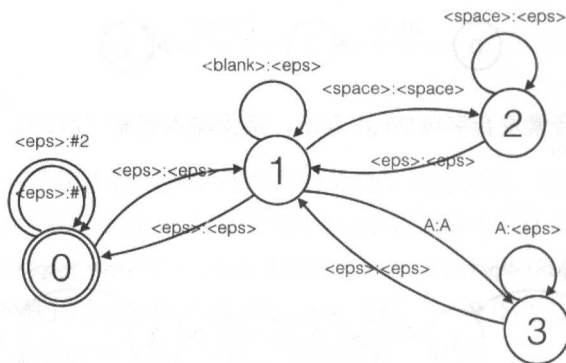


图 16-10 基于字符的 Token WFST 示意图

16.5 Deep Speech

2014 年和 2015 年，百度美国硅谷 AI 实验室连续发表了两篇重量级的端到端语音识别论文 *Deep Speech(DS1)* 和 *Deep Speech2(DS2)*，在英语和普通话语音识别上基本都达到了人类的水平。虽然这两篇论文在理论上并没有多少创新，都是使用已有技术，但是其中介绍了很多工程上的实现细节，而且还涉及了线上部署的经验，这些工程细节和部署经验都是难能可贵的。

传统语音识别的流程特别长，其中包含特征提取、声学建模、发音字典、语言模型等，因此开发一个新的语音识别系统非常困难，而且传统语音识别还需要大量烦琐的手工处理环节，例如背景噪声过滤、回声处理、说话人自适应、构建发音字典等。*Deep Speech* 是一种端到端的语音识别系统，它省去了很多手工处理的中间环节，使得我们很容易处理各种噪声、口音和不同语言等情况，同一套深度学习语音识别框架可以同时应用到英语和普通话的识别任务上。*DS2* 在 *DS1* 的工作上做了很多改进，识别精度和速度都有很大的提升，因此这里着重介绍 *DS2*。

1. 模型

DS2 的模型框架如图 16-11 所示，其中输入直接使用了短时傅里叶变换后的音谱图 (Spectrogram)，传统语音识别 GMM-HMM 的输入是音谱图经过各种变换后得到的特征，

例如 MFCC, PLP。直接使用音谱图, 可以尽可能保留原始语音大量的信息。输入层后面接入的是一些一维或二维的卷积层, 可以把音谱图看成一张二维的图片, 其中一维是时间维度, 另一维是频谱值, t 时刻第 l 层第 i 个激活值为:

$$h_{t,i}^l = f(w_i^l \circ h_{t-c:t+c}^{l-1})$$

其中, c 为上下文窗口大小, \circ 表示卷积操作, f 表示非线性函数, 在 DS2 里面使用的是裁剪后的 ReLU 函数 $f(x) = \min\{\max\{x, 0\}, 20\}$ 。为了缩短时间上的步数, 还经常在卷积层中使用跨步为 s 的卷积操作。

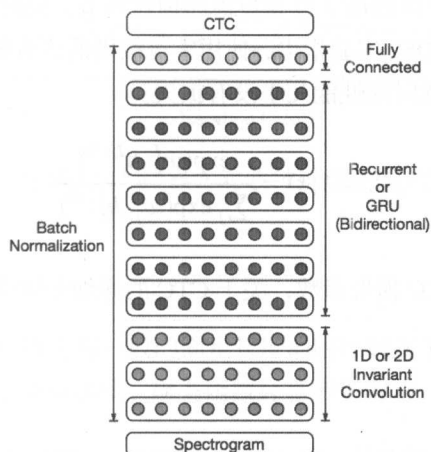


图 16-11 DS2 的模型框架^[8]

在卷积层后面接一个或者多个层叠的双向循环层, 第 l 层 t 时刻的前向输出值 \overrightarrow{h}_t^l 和后向输出值 \overleftarrow{h}_t^l 分别为:

$$\begin{aligned}\overrightarrow{h}_t^l &= g(h_t^{l-1}, \overrightarrow{h}_{t-1}^l) \\ \overleftarrow{h}_t^l &= g(h_t^{l-1}, \overleftarrow{h}_{t+1}^l)\end{aligned}$$

第 l 层总的输出值为前向输出值和后向输出值的加和, 即 $h_l = \overrightarrow{h}_l^l + \overleftarrow{h}_l^l$, 其中激活函数 g 可以为标准的 RNN 循环操作 $\overrightarrow{h}_t^l = f(W^l h_t^{l-1} + \overline{U}^l \overrightarrow{h}_{t-1}^l)$, 也可以是更加复杂的 LSTM、GRU 循环操作。

在循环层后面接一个或者多个全连接层：

$$h_t^l = f(W^l h_t^{l-1} + b^l)$$

最后一层输出层 L 采用 Softmax 输出各个字符的概率值。DS2 在英语中采用了字素 (Graphemic) 级别的输出, 因为 RNN 非常善于拼写, 在大多数情况下不需要额外的自然语言约束就能输出很好的结果, 这样可以省去基于音素的传统语音识别发音字典的构建环节, $l_t \in \{a, b, c, \dots, z, \text{space}, \text{apostrophe}, \text{blank}\}$, 其中 space 为英语单词与单词之间的空格, apostrophe 为英文里面的撇号, blank 为 CTC 中的特殊字符。在普通话识别中, 以往是针对声韵母建模或者短语建模的, 而 DS2 直接采用了字符级别的输出, 其中包含汉字和罗马字符, 总共近 6000 个, 这样就省去声韵母字典和短语的构建环节, 同时使得基于字符级别的语言模型的构建简单不少。而且 DS2 实验指出, 使用字符级别的语音模型, 使用 Beam Search 解码时, 在 Beam 大小为 200 的时候性能几乎没有损失。

$$p(l_t = k|x) = \frac{\exp(w_k^L \cdot h_t^{L-1})}{\sum_j \exp(w_j^L \cdot h_t^{L-1})}$$

最终训练目标是降低 CTC 损失函数, 关于 CTC 的详细介绍请参考第 17 章。

2. 批量规范化

本书第 23 章介绍的批量规范化 (Batch Normalization, BN) 操作, 可以有效地减少内部协变量迁移 (Internal Covariate Shift), 加快训练的收敛速度。在 DS2 深度模型中包含了前向反馈网络、卷积网络和循环网络, 下面我们分别介绍在 DS2 中是如何在这三种不同类型的网络中应用 BN 操作的, 其中 BN 操作为 $\mathcal{B}(x) = \gamma \frac{x - E[x]}{(Var[x] + \epsilon)^{\frac{1}{2}}} + \beta$ 。

(1) 前向反馈网络: 在批量数据集上, 对网络中的每个隐节点激活前做 BN, 即将 $f(W h + b)$ 替换成 $f(\mathcal{B}(W h))$, 每个隐节点都有自己的 γ, β 参数, 因此每一层有多少个隐节点, 就有多少套 γ, β 参数。

(2) 卷积网络: 在批量数据集上, 对每一层的每个特征图 (Feature Map) 上的所有隐节点激活前一起做 BN, 每个特征图内部都使用同样的 γ, β 参数, 因此每一层有多少个特征图 (通常也称为通道 (Channel)), 就有多少套 γ, β 参数。

(3) 循环网络: 在批量数据集上, 每一层对下一层的所有时序隐节点做 BN, 即 $\overrightarrow{h_t^l} = f(\mathcal{B}(W^l h_t^{l-1} + \overrightarrow{U^l h_{t-1}^l}))$, 因此每个循环层都有两套 γ, β 参数。

BN 操作在训练时很好开展，但是在线上真实环境中语音可能需要一个个做识别，这时候在单条语音上做 BN 误差会非常大。解决这个问题的一般做法是保存训练集的均值和方差的平均值，然后在线上直接加载使用。图 16-12 展示了在两种不同类型的 DS2 网络上使用 BN，都能加快训练的收敛速度。

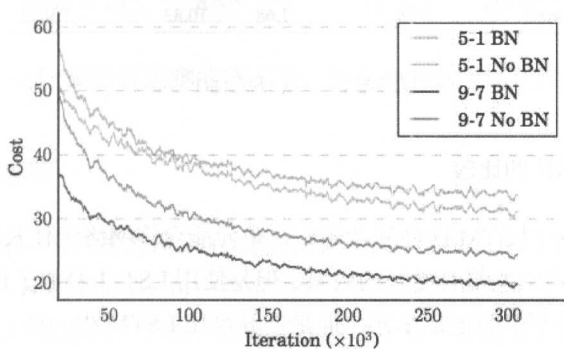


图 16-12 BN 操作对收敛速度的影响^[8]

3. SortaGrad

在变长的时序数据上训练比在一般的固定长度的时序数据上训练要困难很多，一个批量数据集里面音频有长有短，会给梯度、损失函数计算带来困难，一般有两种方法来处理数据变长问题。

(1) 对短的音频数据后面使用静音做补齐，因为 CTC 中的 blank 会吸收这些静音，这样同一个批量数据集中的所有音频长度都相同了，能充分利用一些数学计算库对批量数据的优化。

(2) 像第 1 种方法那样使用静音做补齐，但是在实际计算梯度、损失时使用掩码把这些补齐的位置掩盖掉。这种方法比较精确，但是代码编写比较复杂，一般使用第 1 种方法就足够了。

使用 RNN 模型，对于比较长的语音在训练初期可能会遇到梯度爆炸和梯度消失的问题。梯度消失的问题可以使用 LSTM 或 GRU 这种带门的复杂记忆单元来解决，梯度爆炸的问题一般使用裁剪来解决，还可以使用 TBTT 训练方式来解决，但是 TBTT 可能会影响模型长记忆的能力。一般来说，在训练初期，语音长度越长，梯度爆炸的风险越大。为了试图解决梯度爆炸的问题，得到更好的训练效果，DS2 采用了一种称为 SortaGrad 的启发式方法来训练，在训练的第一轮，按照批量数据中最长语音的长度的升序来进行训练，后面的轮次就采用随

机挑选批量数据的方法。如图 16-13 所示, SortaGrad 在使用 BN 和未使用 BN 的网络中, 都能带来效果的提升, 对未使用 BN 的网络效果更加明显。

	Train		Dev	
	Baseline	BatchNorm	Baseline	BatchNorm
Not Sorted	10.71	8.04	11.96	9.78
Sorted	8.76	7.68	10.83	9.52

图 16-13 SortaGrad^[8]

4. 简单的 RNN 和 GRU 的比较

相比于简单的 RNN, LSTM 这种复杂的记忆单元能允许网络记住长时间的状态, 在很多应用场景中相比简单的 RNN 都有更好的效果, 但是使用 LSTM 增加了很多计算量。而 GRU 是一种效果跟 LSTM 差不多的记忆单元, 但是计算量比 LSTM 少一些:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\tilde{h}_t = f(W_h x_t + r_t \circ U_h h_{t-1} + b_h)$$

$$h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t$$

其中, z_t, r_t 分别代表更新门和重置门, 激活函数 f 使用裁剪的 ReLU。图 16-14 展示了不同类型的 DS2 模型, 使用简单的 RNN 和 GRU 的性能, 可以看出 GRU 相比于简单的 RNN 效果好很多。

Architecture	Simple RNN	GRU
5 layers, 1 Recurrent	14.40	10.53
5 layers, 3 Recurrent	10.56	8.00
7 layers, 5 Recurrent	9.78	7.79
9 layers, 7 Recurrent	9.52	8.19

图 16-14 简单的 RNN 和复杂的 GRU 的效果比较^[8]

但是图 16-15 给出了在相同的计算资源固定的情况下, 7 层的简单 RNN 加上 3 层的二维卷积的效果比 3 层的 GRU 加上 1 层的二维卷积的效果还要好一点。其实这也是可以理解的, 在资源固定的情况下, 简单的 RNN 可以拥有更多的神经单元, 模型的非线性能力更强。当然, 如果在计算资源不受限的情况下, LSTM、GRU 的效果一般会比简单的 RNN 好不少。

Model size	Model type	Regular Dev	Noisy Dev
18×10^6	GRU	10.59	21.38
38×10^6	GRU	9.06	17.07
70×10^6	GRU	8.54	15.98
70×10^6	RNN	8.44	15.09
100×10^6	GRU	7.78	14.17
100×10^6	RNN	7.73	13.06

图 16-15 在计算资源固定的情况下, 简单的 RNN 和 GRU 的效果比较^[8]

5. 时序卷积和频域卷积

20 世纪 80 年代末, 时延神经网络 (TDNN)^[10] 就已经把时序卷积应用到了语音识别上。前面介绍的 CD-DNN-HMM 模型, 每一帧都用到了前后固定窗口的上下文特征, 其实也可以看成一种跨步为 1 的一维卷积, 这些卷积都是在时序上的一维卷积, 效果有限。目前流行的是在时序和频率两个维度的二维卷积, 就是把音谱图当成一种普通的二维图像, 其中长、宽分别为语音时长和频率数, 然后在图像上的所有卷积操作和技巧都可以无缝地迁移过来。

图 16-16 展示了在 DS2 实验中不同层数的一维卷积和二维卷积的效果, 不管是一维卷积还是二维卷积, 卷积层数越多, 一般效果越好, 二维卷积的效果要比一维卷积好, 特别是在噪声数据集上, 二维卷积相比于一维卷积 WER 降了 23.9%, 说明二维卷积对噪声更加鲁棒。

Architecture	Channels	Filter dimension	Stride	Regular Dev	Noisy Dev
1-layer 1D	1280	11	2	9.52	19.36
2-layer 1D	640, 640	5, 5	1, 2	9.67	19.21
3-layer 1D	512, 512, 512	5, 5, 5	1, 1, 2	9.20	20.22
1-layer 2D	32	41×11	2×2	8.94	16.22
2-layer 2D	32, 32	41×11, 21×11	2×2, 2×1	9.06	15.71
3-layer 2D	32, 32, 96	41×11, 21×11, 21×11	2×2, 2×1, 2×1	8.61	14.74

图 16-16 一维卷积和二维卷积^[8]

音谱图是由加窗短时傅里叶变换得到的, 一般间隔为 10ms, 窗口大小为 25ms, 一个 16kHz 的 5s 语音能得到 500 帧, 每一帧包含 201 个不同频率的频谱能量值, 对该音谱图做卷积操作相当于对一张 500×201 的特殊图像做卷积操作, 使用 GPU 卷积操作可以在时间维度上并行进行, 速度非常快, 但是卷积后面的循环层是按照时间维度顺序展开的, 因此不能很好地进行并行化。在 DS2 中为了减少循环层在时间维度上的计算次数, 在其中一个卷积层使用了跨度为 2 的卷积操作, 这样 RNN 层可以减少一半的计算量和存储。

在 DS2 中英语识别的输出是字符级别的, 一个长度为 L 的字符序列, 为了正确输出, 至

少需要 L 帧。据统计, 英语语音每秒平均字符有 14.1 个, 而汉语平均只有 3.3 个, 如果使用前面介绍的跨度为 2 的卷积操作, 那么在语速特别快的情况下, 在英语中可能会出现平均每个字符对应不到一帧的情况, 这会使得 CTC 损失函数计算错误。为了解决这个问题, 在 DS2 中同时使用了非重叠的 bigram 和 unigram 输出: 在字符前后每两个拼接在一起使用 bigram 输出, 但是在含有奇数个字符的单词边界使用 unigram 输出。例如 "the cat sat", 使用单字符级别输出的话, 有 11 个字符 [t h e space c a t space s a t]; 如果使用非重叠的 bigram 和 unigram 输出, 那么就减少到了 8 个字符 [t h e space c a t space s a t]。图 16-17 展示了 bigram 和 unigram 使用不同的跨度、有无语言模型的 WER, 可以看出跨步为 4 的时候性能几乎没有损失, 但是却可以减少 4 倍的 RNN 计算量。

Stride	Dev no LM		Dev LM	
	Unigrams	Bigrams	Unigrams	Bigrams
2	14.93	14.56	9.52	9.66
3	15.01	15.60	9.65	10.06
4	18.86	14.84	11.92	9.93

图 16-17 卷积采用不同跨度的效果比较^[8]

一般来说, 双向 RNN 的效果要比单向 RNN 的效果好, 因为它能够同时利用前后两个方向的信息。但是双向 RNN 在线上环境中使用时就会有问题, 特别是进行流式语音处理时, 系统不可能知道用户要连续说多长时间, 不可能等用户全部说完了再进行解码, 因此这种方式不可取。为了解决这个问题, DS2 在每个单向 RNN 层上面插入了特殊的称为行卷积 (Row Convolution) 的层, 在当前时间点只使用未来少量的信息, 就能达到效果跟双向 RNN 差不多的效果, 如图 16-18 所示。

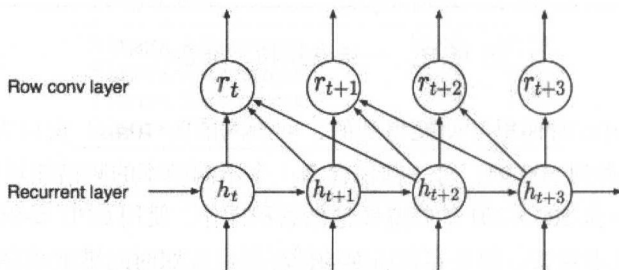


图 16-18 行卷积^[8]

假设我们使用 τ 步的未来信息, 那么行卷积使用的特征矩阵为 $\mathbf{h}_{t:t+\tau} = [h_t, h_{t+1}, \dots, h_{t+\tau}]$,

参数矩阵 \mathbf{W} 的大小跟特征矩阵一样, t 时刻的行卷积值 r_t 按如下方式进行卷积计算:

$$r_{t,i} = \sum_{j=1}^{\tau+1} \mathbf{W}_{i,j} \mathbf{h}_{t+j-1}, \quad 1 \leq i \leq d$$

6. 语言模型

DS2 训练使用了百万条级别的语音语料, RNN 有能力学到一些语言模型, 而且其中最好的模型还非常善于拼写, 在没有语言模型的情况下也能达到不错的效果。但是标注语料相比于正常标准的语言文本语料来说, 量还是非常少的, 因此训练一个额外的语言模型还是有帮助的。DS2 使用 KenLM 来训练平滑的语言模型, 英语使用 2.5 亿条文本, 包含了最常见的 40 万个单词, 最后产生 8.5 亿个 5-gram; 汉语使用百度内部的 80 亿条文本, 最后产生 20 亿个 5-gram。

解码阶段 Beam Search 的句子的打分模型使用了 CTC 对数似然、语言模型和词插入项的线性组合:

$$O(y) = \log(p_{\text{ctc}}(y|x)) + \alpha \log(p_{\text{lm}}(y)) + \beta \text{word_count}(y)$$

其中, α 控制语言模型得分的权重, β 鼓励更多的词出现, 这些值的调优都是在验证集上做的。图 16-19 显示了英语和汉语在有无语言模型情况下的 WER, 可以看出语言模型还是起了很大的作用的。

Language	Architecture	Dev no LM	Dev LM
English	5-layer, 1 RNN	27.79	14.39
English	9-layer, 7 RNN	14.93	9.52
Mandarin	5-layer, 1 RNN	9.80	7.13
Mandarin	9-layer, 7 RNN	7.55	5.81

图 16-19 英语、汉语在有无语言模型情况下的效果比较^[8]

7. 系统优化

DS2 中的深度模型结构复杂, 包含了上千万个参数, 最后使用了上万小时的训练数据, 完成一次完整的训练可能需要很长的时间, 因此 DS2 采用了大量的系统优化技巧来加速训练。

(1) 使用数据并行, 批量大小为 512, 8 块 GPU 同时训练, 训练采用的是同步的 SGD, 即在每个 GPU 上算完 64 个样本的梯度后做一次梯度全局同步, 梯度全局同步使用的是自己实现的环算法, 充分利用了 GPU 直连技术, 速度比使用 OpenMPI 快 2.5 倍。

(2) CTC 前向、后向计算复杂度为 $O(TL)$ ，实现了 GPU 版的 CTC 并行计算，可以减少 10%~20% 的训练时间。

(3) 在 CPU 和 GPU 上实现了类似于 jemalloc 中的 buddy 内存分配算法，预先分配大块的存储空间，减少了频繁动态申请内存的请求次数。

(4) 在 DS1 中还提到了使用模型并行来加速训练，按时间把语音切分成前后两块，前面的卷积层在时间上可以独立并行计算，但是循环层由于 RNN 按时间展开的性质，前后的隐层计算有时间前后的依赖性，一种朴素的做法是直接使用两块 GPU 分别进行前向和后向计算，最后合并双向数据时需要传输大量的数据。DS1 采用了如下高效的做法：两块 GPU 同时进行前向和后向计算，其中一块 GPU 只负责前半部分时间的计算，另一块 GPU 负责后半部分的计算，只需要在中间时间点处交换少量的数据。

(5) 很多真实的线上语音识别场景都要求是实时的，但是 DS2 中包含了双向 RNN、BN、Beam Search 等特别耗费计算资源的操作，同时线上可能有多个人的流式语音请求，如果每次只处理单个的请求，那么 GPU 的利用率、缓存命中率都不高，因此 DS2 线上采用了批量派送 (Batch Dispatch) 的方式，每次把多个用户的流数据合并到一起做批量前向操作，这样会非常的高效。图 16-20 显示了一块 NVIDIA Quadro K1200 GPU 在不同数量的并发流数据下的延迟情况，可以看出同时处理 30 个请求时，在 98% 的情况下延迟不到 200ms，这种延迟对用户的体验毫无影响。

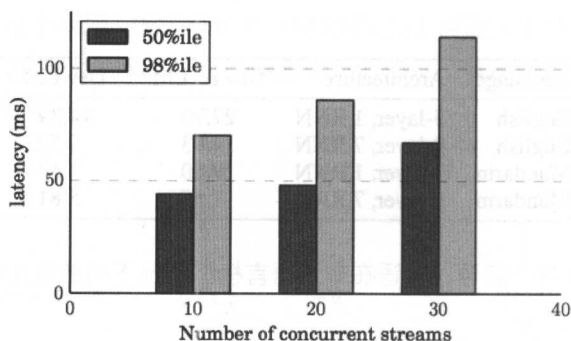


图 16-20 并发流数据延迟情况^[8]

8. 数据集构建与数据增强

语音数据标注成本非常高，英语的一些不同的 native 公开数据集加起来也就 4000 小时左右，如果只使用 native 语料进行训练，那么在非 native 数据上的表现会比较差。网上存在

大量的非 native 数据，但是这些数据文本噪声比较大，而且都比较大，从几分钟到几小时不等。特别长的语音模型输出的中间结果一般在 GPU 显存里面是存不下的，而且长语音标签出错的概率比较大，因此在训练之前需要对这些长语音进行切分。

带 CTC 损失函数的单向 RNN 模型进行 Viterbi 强制对齐很有可能出现一些时延，但是线下可以使用带 CTC 损失函数的双向 RNN 来做强制对齐，而且对齐还比较精确，对齐之后根据 blank 连续出现的次数对数据做切分，那些出现连续 blank 较长的地方很有可能是句子结束的地方。切分完之后，再提取一些特征，训练一个二分类模型把一些质量较差的对齐语音段去除。

噪声非常影响模型的识别精度，一般公开的含有噪声的数据集噪声种类比较少，而且从录音棚里面收集训练数据都是在安静环境下进行的，但是在现实生活环境中各种各样的噪声，模型在没有见过的噪声中识别，效果一般都比较差。这就需要我們使用数据增强技术来对现有的数据集加入各个特征，噪声合成一般都采用叠加技术：

$$\hat{x}^{(i)} = x^{(i)} + \xi_1^{(i)} + \xi_2^{(i)} + \dots$$

其中， $x^{(i)}$ 为原始声音， $\xi_1^{(i)}, \xi_2^{(i)}$ 为不同类型的噪声。

16.6 Chain

在 *Interspeech 2016*, Kaldi 开源作者 Povey 博士发表了一篇 Lattice-free MMI (LF-MMI) 判别式训练的论文^[9]，该论文在 Kaldi 里面的实现称为 Chain 模型。传统的序列判别式训练首先需要 GMM-HMM 做强制对齐，然后使用 CD-DNN-HMM 双向模型进行帧级别交叉熵 (CE) 损失函数的预训练，最后使用判别式损失函数进行调优。该论文中提出的不需要帧级别交叉熵预训练的序列判别式神经网络训练方法，跟传统的模型有如下几点不同。

(1) 在 NN 输出层使用 3 倍小的帧率，这样可以显著地减少预测的时间，使得线上实时解码更加容易。

(2) 由于使用了小帧率，因此使用跟传统 3 状态 HMM 不一样的 HMM，每个音素的 HMM 可以只经过一个状态。

(3) 固定 HMM 的转移概率，而且不再更新，因为在 NN 模型输出值中包含转移概率。

(4) 训练直接使用跟 CTC 一样的序列目标函数：最大化正确文本序列的条件似然 $\log(p(W|O))$ ，但区别是 CTC 是做局部归一化的，而 MMI 是做全局归一化的。

(5) 在 GPU 上实现 Lattice-free MMI 训练, 直接在解码图上做完整的前向-后向, 解码图是基于音素级别的 n -gram。

1. LF-MMI

在前面章节中我们介绍了最大互信息判别式训练准则, 其建模使得观察变量和词序列之间的互信息最大。

$$O_{\text{MMI}}(\theta) = \sum_{r=1}^R \log \frac{p(O_r|W_r, \theta)^\kappa p(W_r)}{\sum_W p(O_r|W, \theta)^\kappa p(W)}$$

传统 GMM-HMM 使用的 ML 准则只考虑为给定正确文本的似然 $p(O_r|W_r, \theta)$ 建模, 而 MMI 准则除为正确文本建模之外, 还考虑与正确文本相似的其他文本的似然, 尽量提高正确文本的似然, 压低竞争对手的似然, 使得正确文本的概率值相对于相似文本的概率值要有一个区分性。

MMI 判别式训练不但对传统 GMM-HMM 模型有效, 对 CD-DNN-HMM 模型也同样有效, Povey 博士在 2013 年发表的一篇论文^[3]中详细介绍了在 CD-DNN-HMM 中如何使用各种判别式训练准则, 在前面章节中我们对判别式训练的求导进行了详细推导, 梯度公式中的状态占有后验概率 $\gamma^{\text{num}}(x_{ij}^r), \gamma^{\text{den}}(x_{ij}^r)$ 需要使用词级别的分子、分母 Lattice 才能进行高效的计算, 词 Lattice 是在解码的时候结合语言模型构建的, 由于词一般比较多, 所以最后构建的 Lattice 都比较大, 花费的时间也比较长。而且在后验概率的求解中使用了词级别和音素级别的前向-后向算法, 逻辑比较复杂, 非常不利于 GPU 这种单指令多数据的并行计算模式, 最关键的是基于词级别的 Lattice 需要在统一 WFST 上解码构建词图, 大规模连续语音识别的统一 WFST 会占用很多内存, GPU 基本不可能存放下统一 WFST。

借助 CTC 的思想, Chain 模型直接训练 MMI 序列目标函数, 不需要帧级别的 CE 做初始化, 整个训练过程跟传统的 MMI 非常相似, 对 Softmax 输入的求导就等于分子的状态占有后验概率 $\gamma^{\text{num}}(x_{ij}^r)$ 和分母的状态占有后验概率 $\gamma^{\text{den}}(x_{ij}^r)$ 的差。但是有一个很大不同的地方就是 Chain 模型计算后验概率不使用词 Lattice, 而是直接使用音素 n -gram, 构建音素级别的解码图。LF-MMI 中分母的状态占有后验概率 $\gamma^{\text{den}}(x_{ij}^r)$ 直接基于音素级别的解码图做前向-后向, 而分子的状态占有后验概率 $\gamma^{\text{num}}(x_{ij}^r)$ 是在给定的文本上做前向-后向的。还有一点不同, Chain 模型的 DNN 直接输出伪似然, 声学模型不需要除状态先验概率 $p(x)$ 。

2. HMM 拓扑结构

传统的 HMM 拓扑结构是含有 3 状态的从左到右的 HMM, 其中每个状态都必须经过一次, 也就是说, 每个音素至少对应着 3 帧; 但是 Chain 模型在输出层采用了 3 倍小的帧率,

因此继续使用 3 状态的 HMM 可能会在语速特别快的语音中出现帧数不够的情况，Chain 采用如图 16-21 所示的单 HMM 拓扑结构。

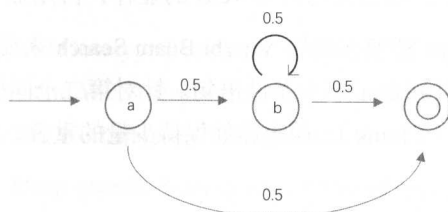


图 16-21 Chain 采用单状态 HMM

其中，状态 a 必须经过且只经过一次，状态 a 可以转移到状态 b，也可以直接跳到结束状态，状态 b 可以自旋，因此，可能会出现如下状态序列：a, ab, abb, abbb, …，可以把状态 b 当成 CTC 中的 blank，与 CTC 不同的地方是，Chain 模型的每个三音素 HMM 都有属于自己的状态 b。

3. 分母、分子 FST

Chain 模型虽然是词 Lattice-free 的，但是为了计算分母还需要构建解码图，只不过这时候不是基于词级别的计算，而是基于更细粒度的音素级别构建解码图的。如果不考虑语言模型，那么计算任意时刻任意状态的后验概率，只需要跑一个普通的前向-后向算法即可计算出所有状态序列下的后验概率。由于我们是为序列建模的，序列先后的依赖概率可以部分通过声学模型建模出来，但是语音训练数据量级相对于文本语料来说还是太小了，所以计算后验概率时需要让语言模型参与进来。在 Chain 模型中使用了 4-gram 的音素语言模型，是直接来自训练数据中做强制对齐统计出来的，为了减少语言模型 WFST 的边的数量，这里没有使用回退平滑操作。

音素级别的解码图 WFST，可以直接通过 HMM 的 WFST— H 、上下文依赖的三音素 WFST— C 、音素语言模型 WFST— P 组合 $N = H \cdot C \cdot P$ ，当然中间还要做一些 WFST 的优化操作。在 Switchboard-1 数据集上，最终音素解码图包含大约 2.4 万个状态、22 万个转移。一般来说，状态数量决定内存的占用，转移数量决定计算量，相比于词级别的解码图，音素级别的解码图要小很多，而且分母解码图是所有数据共享的。

直接在分母解码图上执行完全的前向-后向算法会非常慢，所以必须使用 GPU 进行并行计算。但是使用 GPU 进行计算会遇到如下两个问题。

(1) Beam Search 代码中包含大量的条件语句，使得 GPU 会串行遍历所有的条件路径，导致最终的并行效率并不高效。

(2) 训练时一般都是在小批量数据上计算梯度的, 在 SWB 数据集上分母解码图大约有 30000 个状态, 如果训练批量大小为 128, 平均语音长度为 6s, 采用 3 倍小的帧率, 前向计算采用单精度浮点数表示, 那么大约需要 4GB 的显存, 占用显存特别大。

针对第一个问题, Chain 模型不使用 Viterbi Beam Search 来做近似计算, 而是直接执行完全的前向-后向算法, 这样减少了不少条件语句。针对第二个问题, Chain 模型把语音切分成 1s 或 1.5s 固定长度的块 (Chunk), 在边界处保持少量的重叠, 而且尽量保持小的解码图, 这样就解决了显存问题。

分子中的状态占有后验概率是在给定文本下进行计算的, 与分母一样, 我们也创建一个给定文本的分子 WFST, 最后计算前后-后向就非常简单了。但是会受到 GPU 的显存限制, 因此使用以上提到的分块方法, 对分子解码图进行分裂。为了对分子解码图做合适的分裂, 需要在对齐的时候加一些时间约束, 具体细节请参考文献 [9]。

4. 实现效果

Chain 模型论文^[9]中的实验结果表明, LF-MMI 模型在各种数据集上都比传统的基于 Lattice 的判别式训练效果好, 而且由于 HMM 状态数量减少了, 模型大小也减少不少, 这对于把模型移植到手机等移动设备上有很大的利好。最关键的是输出采用 3 倍小的帧率, 训练速度有 5~10 倍的提升, 解码速度有 3 倍的提升。在我们自己的实验中, 基于 Chain 模型完全可以胜任实时流式语音识别的任务, 可以说 Lattice-free 给语音识别提供了一个全新的方向, 是一项具有里程碑意义的工作。

参考文献

- [1] Dahl, George E., et al. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing* 20.1 (2012): 30-42.
- [2] Hinton, Geoffrey, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29.6 (2012): 82-97.
- [3] Veselý, Karel, et al. Sequence-discriminative training of deep neural networks. *Interspeech*. 2013.
- [4] Peddinti, Vijayaditya, Daniel Povey, and Sanjeev Khudanpur. A time delay neural network architecture for efficient modeling of long temporal contexts. *INTERSPEECH*. 2015.

[5] Graves, Alex, and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 2014.

[6] Miao, Yajie, Mohammad Gowayyed, and Florian Metze. EESSEN: End-to-end speech recognition using deep RNN models and WFST-based decoding. *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*. IEEE, 2015.

[7] Hannun, Awni, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[8] Amodei, Dario, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *International Conference on Machine Learning*. 2016.

[9] Povey, Daniel, et al. Purely Sequence-Trained Neural Networks for ASR Based on Lattice-Free MMI. *INTERSPEECH*. 2016.

[10] Waibel, Alex, et al. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing* 37.3 (1989): 328-339.

[11] Kenny, Patrick, Gilles Boulianne, and Pierre Dumouchel. Eigenvoice modeling with sparse training data. *IEEE transactions on speech and audio processing* 13.3 (2005): 345-354.

[12] Dehak, Najim, et al. Front-end factor analysis for speaker verification. *IEEE Transactions on Audio, Speech, and Language Processing* 19.4 (2011): 788-798.

[13] Yu, Dong, and Li Deng. *Automatic speech recognition: A deep learning approach*. Springer, 2014.

17

CTC 解码

第 5 章提到 RNN 相比 CNN 更适合序列类型的数据，因而在语音识别领域有广泛应用，但序列的识别往往还存在一个对齐解码的问题，RNN 需要一个额外搭档来解决这个问题。在语音识别方面，CTC（Connectionist Temporal Classification）解码是 RNN 的完美搭档之一，本章将具体介绍 CTC 解码的原理。

17.1 序列标注

序列标注（Sequence Labelling）的目标是给一个输入序列赋予一个标注序列，其中标注序列中的每个元素来自于一个固定的字符集合。如图 17-1 所示，给定一段语音，需要识别出语音对应的文字；给定一串手机拍照的印刷体文本，需要识别出对应的文字；给定一张手写的文字图片，需要识别出其中的文字串；给定一个手写公式，需要识别出其中的公式内容。其他的，如看图说话、音乐生成、聊天机器人、视频分析、机器翻译等都涉及序列标注的问题。



(a) 语音识别



(c) 手写英语识别



(b) 印刷体识别



(d) 手写公式识别

图 17-1 序列标注示例

17.2 序列标注任务的解决办法

序列标注任务一般有三种解决办法。

- (1) 序列分类 (Sequence Classification): 每一个输入序列都被赋值一个单独的类。
- (2) 分割分类 (Segment Classification): 每一个输出序列都被切分为多段, 每段对一个分类标注。
- (3) 时序分类 (Temporal Classification): 任何输入段和输出标签之间的对齐关系都是允许的。

三者之间的关系如图 17-2 所示。可以看出, 序列分类是一种特殊的分割分类, 即整个输入序列被切分为一段; 分割分类是一种特殊的时序分类, 即对齐关系要求与切分对应。



图 17-2 序列分类、分割分类与时序分类的关系

17.2.1 序列分类

如果把输入序列以及标注的标签分别看作整体, 那么对应的序列标注问题就是序列分类 (Sequence Classification) 问题, 因为每一个输入序列都被赋值一个单独的类。序列分类技术可以用于人脸识别 (判断一整张图片是不是一个人脸) 和手写英语单词识别 (判断一张图片中的内容对应的是哪个英语单词, 输入是一个单词的手写图片, 输出是对应的某个单词, 不细化到字母) 等, 比如 MNIST 对应的手写识别问题就是一个序列分类问题。

由于分类是针对整个输入序列进行的, 类似于传统的分类问题, 序列错误率 (Sequence Error Rate) $E^{\text{seq}}(h, S')$ 表示如下:

$$E^{\text{seq}}(h, S') = \frac{1}{S'} \sum_{(x,z) \in S'} \begin{cases} 0, & h(x) = z \\ 1, & \text{其他} \end{cases}$$

其中 h 为序列分类模型: $\mathcal{X} \mapsto \mathcal{Z}$, \mathcal{X} 是输入空间, 即所有可能的输入序列; \mathcal{Z} 是输出空间, 即所有可能的输出标注序列; $h(x)$ 则为 h 模型针对输入序列 x 的预测值。 S 是训练集, S' 是测试集, S' 中的每条数据都是一个序列对 (x, z) , 其中 x 是输入序列 (x_1, x_2, \dots, x_T) , z 是目标序列 (标注序列) (z_1, z_2, \dots, z_U) 。一般来说, 输入序列的长度是大于或等于输出序列的长度的。

17.2.2 分割分类

序列分类比较适合对英语单词等的识别, 因为单词是一个天然的整体, 字母之间的空隙很小, 而单词之间的空格比较明显; 但是对汉字的识别则不一样, 字与字之间的间隙很小。如图 17-1 中所示的印刷体识别, 如果把整个文本串作为输入, 那么输入空间就会非常大, 所需要的标注数据总量也会变得特别多。假设只考虑常用的 3500 个左右的汉字, 每行平均 20 个汉字, 那么输入空间的大小为 3500^{20} 。而对于英语单词的识别, 假设考虑 10000 个常见单词, 每个单词整体作为输入, 那么输入空间也只有 10000 这个量级, 远小于一行汉字的输入空间。如果要降低汉字识别的输入空间大小, 一种非常自然的方式就是引入分割, 将一行文本切分为单个的汉字, 然后针对每个汉字进行识别, 这样输入空间就可下降为汉字总数级别。这种把输入图像切分为多个小段或者小块的方法一般也称为图像切分。这种先分割再识别的方法称作分割分类 (Segment Classification)。

一般定义分割错误率 $E^{\text{seg}}(h, S')$ 为:

$$E^{\text{seg}}(h, S') = \frac{1}{Z} \sum_{(x, z) \in S'} \text{HD}(h(x), z)$$

其中 Z 是测试集中目标序列的总长度, 即:

$$Z = \sum_{(x, z) \in S'} |z|$$

$\text{HD}(h(x), z)$ 是等长序列 $h(x)$ 与 z 之间的汉明距离 (Hamming Distance), 即两者对应位置上不同元素的个数。

和序列分类相比, 分割分类对输入序列进行了拆分, 输入空间变小了, 但是也会额外引入分割错误。比如对英语单词的识别除了用序列分类, 也可以考虑用分割分类, 将英语单词切分到字母, 然后进行单个字母的识别, 最后组合成单词。这样我们需要的标注数据的输入空间只有 52 左右 (只考虑大小写字母)。但是因为单词内部的切分很难, 尤其是手写时, 很

多人会连写，这样切分错误率就会很高，因此在实际当中，手写英语识别一般采用序列分类，而中文识别往往采用分割分类。但有时也采取折中方案，比如手写英语识别问题，如果我们并不关心英语单词是什么，而是关心字母级别的 Bi-Gram，这时候就可以考虑把单词切分到两个字母的长度，然后再进行识别。对两个字母本身的识别依然可以看作序列分类，但是考虑前面的切分，总体上应该归属分割分类的范畴。

此外，分割分类相比序列分类也丢失了一些上下文信息，被切分到汉字后，当前汉字和旁边汉字就没有关系了。但在实际当中，语言的上下文信息是非常重要的。为了弥补这个缺点，分割分类一般会配合语言模型等一起使用，即在识别之后结合语言本身的上下文信息进行纠错。

17.2.3 时序分类

很多时候，我们只知道标注序列的长度小于或等于输入序列的长度，但对其位置的对应关系等一无所知，这样的问题是时序分类（Temporal Classification）问题^[1]。语音识别是典型的时序分类问题，输入的语音帧数比目标序列（一般为因素序列或文本序列）的长度要长，而且很难事先知道哪几帧是一起发一个音的，也就是很难直接做分割分类。

时序分类和分割分类的最大区别在于，时序分类要求算法能够决策输入序列中的哪一部分与输出序列的对应片段匹配，也就是存在一个隐式或显式的序列结构模型。

在时序分类问题中，分割错误率是无法获得的，因为输入序列和目标序列之间的对齐关系未知。但是我们可以评估预测序列需要经过多少次插入、删除、替换操作才能变成目标序列（编辑距离），也就是标注错误率 $E^{\text{label}}(h, S')$ ：

$$E^{\text{label}}(h, S') = \frac{1}{Z} \sum_{(x,z) \in S'} \text{ED}(h(x), z)$$

其中 Z 是测试集中目标序列的总长度， $\text{ED}(h(x), z)$ 是序列 $h(x)$ 与 z 之间的编辑距离（Edit Distance），即将 $h(x)$ 变为 z 所需要的插入、删除、替换操作的最少数量。

17.3 隐马模型

隐马尔可夫模型（Hidden Markov Model, HMM，简称“隐马模型”）是序列分类问题的一种选择。隐马模型包含一系列隐藏状态以及依赖于这些状态的观察序列。然而，传统的隐马模型是比较受限的，包括：

(1) 状态集合的大小 n 不可过大, 因为 Viterbi 算法的时间复杂度为 $O(n^2T)$, 其中 T 为时刻数量, 其状态转换矩阵的大小也是 $O(n^2)$ 级别的。

(2) 隐马模型一般采用条件独立假设: $p(q_t|q_{t-1}, q_{t-2}, \dots, q_1) = p(q_t|q_{t-1})$, 即 t 时刻的隐藏状态 q_t 一般只依赖于其相邻的前一状态 q_{t-1} , 即一阶隐马模型。如果需要依赖于前面 k 个状态, 则时间复杂度可能达到 $O(n^kT)$ 级别, 这种指数级增长使得长时间窗口方案往往在性能上不可行。

RNN 则克服了隐马模型的缺点, 能够捕获长时间窗口的依赖, 而这种依赖不会受限于状态空间太大的问题。因为神经网络是一种联结主义模型, 所有的输入都会被组合在一起, 假设存在 n 个输入节点, 即使每个输入节点的状态都是二值的, 该层网络能表示的状态空间也已达 2^n 。如果输入节点的状态可以取浮点数, 其对应的状态空间则会变得更大。当然, 隐马模型与神经网络相结合的混合方法也是一个不错的选择, 这方面的研究也很多^{[2][3]}。

RNN 通过内部节点之间的环形关系来表征序列的时序关系, 近年来, LSTM (Long Short-Term Memory) 在很多任务上都取得了突破性的进展。而 CTC^{[4]~[7]} 则是 RNN 模型用于时序分类任务时理想的解码模型之一, 使用 CTC 不需要对输入序列进行预分割, 也不需要输出序列进行额外的后处理。手写识别和语音识别等相关实验表明, CTC+BLSTM (双向 LSTM) 的网络结构比 HMM+RNN 的混合解决方案更好^[8]。

17.4 CTC 基本定义

利用 RNN 等模型进行时序分类, 不可避免地会出现很多冗余信息, 比如一个字母被连续识别两次, 这就需要一套去冗余的机制, 但是简单地看到两个连续字母就去冗余的方法也会有问题, 因为存在 cook、geek 之类的单词。当然, 这难不倒手握 blank 神器的 CTC。

给定一个序列标注问题, 其输出标注来自于一个字母集合 L , CTC 则是一个 Softmax 输出层, 其标注字符集大小为 $|L| + 1$, 即增加一个空白字符 (blank)。为了方便起见, 后面我们统一记为 blank。也就是每个输出单元要么为 L 中的一个元素, 要么是 blank。 y_k^t 表示 t 时刻输出单元为字符 k 的概率。给定长度为 T 的输入序列 x 以及训练集 S , π 表示元素来自于集合 $L' = L \cup \{\text{blank}\}$ 的长度为 T 的序列集合^[8], 一般 π 称为路径, 其对应的概率为:

$$p(\pi|x, S) = \prod_{t=1}^T y_{\pi_t}^t$$

其中 T 为路径 π 对应的长度, $y_{\pi_t}^t$ 为路径上第 t 个节点为 π_t 的概率。

如图 17-3 所示, 假设时序分类任务是要识别手写英文 Mary。先假设 CTC 的输入 x 长度为 7, RNN 字母集合为 52 个字母及标点符号, 路径 π 为 M_a_r_y (下画线 “_” 表示 blank), 则其对应的概率为:

$$p(\pi = \text{M_a_r_y} | x, S) = \prod_{t=1}^T y_{\pi_t}^t = y_M^1 \cdot y_{\text{blank}}^2 \cdot y_a^3 \cdot y_{\text{blank}}^4 \cdot y_r^5 \cdot y_{\text{blank}}^6 \cdot y_y^7$$

当然, π 也可能是 NN_ar_y 或其他, 取决于前面 RNN 模型的识别效果。如果是一个比较好的识别模型, 我们一般会期望 Mary 的合法路径的概率比 NN_ar_y 之类的更高一些。

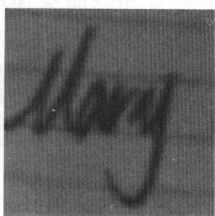


图 17-3 手写识别示例: Mary

这个变种其实可以形式化地定义为一种多对一的映射 $\mathcal{B}: L^T \mapsto L^{\leq T}$, 其中 $L^{\leq T}$ 表示的是在原始的标注字母集 L 中小于或等于长度 T 的序列, 规则很简单 (有先后顺序):

- 相邻的重复字母保留一个。
- 移除所有 blank。

例如 $\mathcal{B}(\text{M_a_r_y}) = \mathcal{B}(\text{M_aa_r_y}) = \mathcal{B}(\text{Maa_r_y}) = \text{Mary}$ 。从直观上说, 相邻的重复字母往往是因为窗口太小导致同一个元素被识别多次, 所以相邻的重复字母只保留一个。但如果目标字符串含有连续的重复字母, 比如 cook、doom 等, 则重复字母之间必须引入 blank, 这样 CTC 才可以正确地解码。例如 $\mathcal{B}(\text{c_o_o_k}) = \text{cook}$, 但是 $\mathcal{B}(\text{c_oo_k}) = \text{cok}$ 。blank 可以避免噪声带来影响, 比如语音识别的因素之间存在停顿或背景噪声、OCR 的字符之间存在背景或噪声。新的输出字母要么出现在 blank 变字母的时候, 要么出现在一个字母变成另一个不一样的字母的时候。这种直观的理解很重要, 可以帮助理解后面的动态规划算法。

进一步, 我们定义 \mathcal{B}^{-1} , 如果路径 π 和原始的字符序列 l 满足 $l = \mathcal{B}(\pi)$, 则 $\pi = \mathcal{B}^{-1}(l)$ 。不难看出 $\mathcal{B}^{-1}(\text{Mary})$ 集合中包含 Mary、M_a_r_y、M_aa_r_y 等元素。直观上, $\mathcal{B}^{-1}(l)$ 相当于原始序列 l 做如下操作 (有先后顺序):

- 插入 blank, 其中 l 中相邻且相同的两个字母之间必须插入 blank。
- 插入相邻的重复字母。

这里需要注意的是，如果 l 中相邻且相同的两个字母之间不插入 **blank**，则会使目标串发生变动。比如在 $l = \text{cook}$ 的两个 **o** 之间不插入 **blank**，则可能变为 c_ook ，但 $\mathcal{B}(c_ook) = \text{cok} \neq \text{cook}$ 。

由于路径之间是互斥的，对于标注序列，其条件概率为所有映射到它的路径概率之和：

$$p(l|x) = \sum_{\pi \in \mathcal{B}^{-1}(l)} p(\pi|x)$$

这种通过映射 \mathcal{B} 和所有候选路径概率之和的方式使得 CTC 不需要对原始的输入序列进行准确的切分，当然这也说明 CTC 不适合那些需要知道准确切分位置的序列标注任务，初次接触 CTC 的读者都需要注意这个问题。

CTC 可以与任意的 RNN 模型搭配，但是考虑到标注概率与整个输入串相关，而不是仅与前面小窗口范围内的片段相关，因此双向的 RNN/LSTM 模型更为合适。

17.5 CTC 前向算法

依然假设 l 为原始的标注序列，现针对 l 进行如下修改：

- 开头和结尾各插入一个 **blank**。
- 每两个字母之间（此处不用考虑是否为重复字母）也插入一个 **blank**。

将修改后的序列记为 l' ，显然 l' 的长度为 $2|l| + 1$ ；反过来， $|l| = \frac{|l'|}{2}$ ，且 l 与 l' 是一一对应的关系。

例如， $l = \text{Mary}$ 时 $l' = _M_a_r_y_$ 。

CTC 针对 l' 的前缀计算路径概率，既允许 **blank** 和非 **blank** 之间的转换，也允许相邻的不同字母之间的变化。对于一个标注序列 l ，我们定义前向变量 $\alpha_t(s)$ 为满足以下条件的所有路径 $\pi_{1:t}$ 的概率之和：

$$\mathcal{B}(\pi_{1:t}) = l_{1:s/2} \text{ 且 } \pi_t = l'_s$$

根据前面的定义可知， l' 的奇数位均为 **blank**，偶数位均为 l 中对应字母 $l_{s/2}$ ，注意这里 s 从 1 开始。比如， $l = \text{Mary}$ 时， $l' = _M_a_r_y_$ ，那么 $l'_{1:3} = _M_$ ， $l_{1:3/2} = M$ ； $l'_{1:4} = _M_a$ ， $l_{1:4/2} = Ma$ 。

前向变量 $\alpha_t(s)$ 的计算公式定义如下:

$$\alpha_t(s) = P(\pi_{1:t}, \mathcal{B}(\pi_{1:t}) = l_{1:\frac{s}{2}}, \pi_t = l'_s | x) = \sum_{\pi: \mathcal{B}(\pi_{1:t}) = l_{1:s/2}} \prod_{l'=1}^t y_{\pi_t}^{l'}$$

利用动态规划的思想, $\alpha_t(s)$ 完全可以由 α_{t-1}^s 、 α_{t-1}^{s-1} 和 α_{t-1}^{s-2} 表示。

有了上面的公式, 我们可以知道 l 的概率可以表示为 $\alpha_T(|l'|)$ ($\pi_T = l'_{|l'|} = \text{blank}$) 和 $\alpha_T(|l'| - 1)$ ($\pi_T = l'_{|l'|-1} = l_{|l|} \neq \text{blank}$) 两个前向概率之和, 其中 $l_{|l|}$ 为 l 的最后一个字母。即:

$$\begin{aligned} p(l|x) &= \alpha_T(|l'|) + \alpha_T(|l'| - 1) \\ &= P\left(\pi_{1:T} : \mathcal{B}(\pi_{1:T}) = l_{1:\frac{|l'|}{2}} = l, \pi_T = l'_{|l'|} = \text{blank} \middle| x\right) + \\ &\quad P\left(\pi_{1:T} : \mathcal{B}(\pi_{1:T}) = l_{1:\frac{|l'|-1}{2}} = l, \pi_T = l'_{|l'|-1} = l_{|l|} \middle| x\right) \end{aligned}$$

假设允许所有路径可以以 l 的第一个字母或者 blank 开始, 我们得到下面的动态规划初始化条件:

$$\begin{aligned} \alpha_t(0) &= 0 \\ \alpha_1(1) &= y_{\text{blank}}^1 \quad // \text{以 blank 开始} \\ \alpha_1(2) &= y_{l_1}^1 \quad // \text{以 } l \text{ 的第一个字母开始} \\ \alpha_1(s) &= 0, \forall s > 2 \quad // \text{其他} \end{aligned}$$

下面我们来推导路径 $\pi_{1:t}$ 前向概率 $\alpha_t(s)$ 的递推公式。

(1) 当 $l'_s = \text{blank}$ 时:

由 l' 的定义可知, s 必然为奇数, 且 $l'_{s-1} \neq \text{blank}$ 。因为 $\alpha_t(s) \Rightarrow \mathcal{B}(\pi_{1:t}) = l_{1:s/2}$, 且 $\pi_t = l'_s = \text{blank}$, 所以 $\pi_{1:t-1}$ 对应的前向概率分为两种情况。

- $\alpha_{t-1}(s-1)$ ($\pi_{t-1} = l'_{s-1} \neq \text{blank}$)
- $\alpha_{t-1}(s)$ ($\pi_{t-1} = l'_s = \text{blank}$)

如果 $\pi_{1:t-1}$ 的前向概率是 $\alpha_{t-1}(s-2)$, 也就是 $\pi_{1:t-1}$ 不包含 l'_{s-1} , 由于 $\pi_t = \text{blank}$ 已知, 且由 $l'_s = \text{blank}$ 可知 $l'_{s-1} \neq \text{blank}$, 因此 $\alpha_t(s)$ 不可满足, 换句话说, 当 $l'_s = \text{blank}$ 时, $\alpha_{t-1}(s-2)$ 无法递推出 $\alpha_t(s)$ 。

因此递推公式为:

$$\alpha_t(s) = y_{l'_s}^t \alpha_{t-1}(s-1) + y_{l'_s}^t \alpha_{t-1}(s)$$

(2) 当 $l'_s \neq \text{blank}$ 时:

由 l' 的定义可知, s 必然为偶数, 且 $l'_{s-1} = \text{blank}$ 。因为 $\alpha_t(s) \Rightarrow \mathcal{B}(\pi_{1:t}) = l_{1:s/2}$, 且 $\pi_t = l'_s \neq \text{blank}$, 所以 $\pi_{1:t-1}$ 对应的前向概率分为两种情况。

- $\alpha_{t-1}(s)$, 即 $\pi_{t-1} = l'_s$, 在这种情况下 $t-1$ 和 t 时刻连续出现了两个相同的 l'_s 且是非空字符, 满足映射 \mathcal{B} 中的重复规则, 此时对应递推概率为: $y_{l'_s}^t \alpha_{t-1}(s)$ 。
- $\alpha_{t-1}(s-1)$, 即 $\pi_{t-1} = l'_{s-1} = \text{blank}$, 如果 $\pi_{1:t-1}$ 的前向概率是 $\alpha_{t-1}(s-2)$, 也就是 $\pi_{1:t-1}$ 不包含 l'_{s-1} (blank), 由 s 为偶数可知, $l'_{s-2} = l_{(s-2)/2} \neq \text{blank}$, 那么能否由 $\alpha_{t-1}(s-2)$ 递推出 $\alpha_t(s)$ 又需要细分为两种情况:
 - $l'_{s-2} = l'_s$, 虽然在 CTC 中 $l'_{s-1} = \text{blank}$ 可以被跳过, 但因为 blank 两端的字母相同, 跳过 blank 会触发重复字母去重规则, 所以在这种情况下, 不可直接由 $\alpha_{t-1}(s-2)$ 递推出 $\alpha_t(s)$ 。
 - 其他, 直接跳过 $l'_{s-1} = \text{blank}$, 当然, 由 $s-2 \geq 1$ 可知 $s > 2$ 。

因此递推公式为:

$$\alpha_t(s) = \begin{cases} y_{l'_s}^t \alpha_{t-1}(s-1) + y_{l'_s}^t \alpha_{t-1}(s), & s \leq 2 \text{ 或者 } l'_{s-2} = l'_s \\ y_{l'_s}^t \alpha_{t-1}(s-2) + y_{l'_s}^t \alpha_{t-1}(s-1) + y_{l'_s}^t \alpha_{t-1}(s), & \text{其他} \end{cases}$$

综合前面的所有公式, 推导出 CTC 前向概率的动态规划递推公式如下:

$$\alpha_t(s) = y_{l'_s}^t \begin{cases} \sum_{i=s-1}^s \alpha_{t-1}(i), & s \leq 2 \text{ 或者 } l'_s = \text{blank} \text{ 或者 } l'_{s-2} = l'_s \\ \sum_{i=s-2}^s \alpha_{t-1}(i), & \text{其他} \end{cases}$$

其中, $\alpha_t(s) = 0, \forall s < |l'| - 2(T-t) - 1$ 。

以上条件对应于图 17-4 中右图上部分路径不通的点^[8]。在图 17-4 中箭头代表前向递推方向, 其中空白圆圈 (blank) 可以跳过 (上下两个字母不同的时候); 而黑色圆圈表示非空字符, 不可跳过, 与递推公式一致。

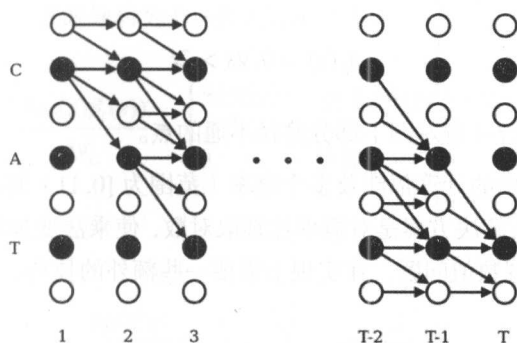


图 17-4 CTC 前向示例

17.6 CTC 后向算法

CTC 后向算法与前向算法类似。后向变量 $\beta_t(s)$ 定义为满足以下条件的所有路径 $\pi_{t+1:T}$ 的概率之和（注意下标是从 $t+1$ 开始的）：

$$\mathcal{B}(\pi_{t:T}) = l_{s/2:|l|} \text{ 且 } \pi_t = l'_s$$

故后向变量 $\beta_t(s)$ 的计算公式如下：

$$\beta_t(s) = P(\pi_{t+1:T} : \mathcal{B}(\pi_{t:T}) = l_{s/2:|l|}, \pi_t = l'_s | x) = \sum_{\pi : \mathcal{B}(\pi_{t:T}) = l_{s/2:|l|}} \prod_{t'=t+1}^T y'_{\pi_{t'}}$$

后向变量的初始化和递推公式如下：

$$\beta_t(|l'| + 1) = 0 \quad \forall t$$

$$\beta_T(|l'|) = 1$$

$$\beta_T(|l'| - 1) = 1$$

$$\beta_T(s) = 0, \forall s < |l'| - 1$$

$$\beta_t(s) = \begin{cases} \sum_{i=s}^{s+1} \alpha_{t+1}(i) y'_{l'_i}, & s \leq 2 \text{ 或者 } l'_s = \text{blank} \text{ 或者 } l'_{s-2} = l'_s \\ \sum_{i=s}^{s+2} \alpha_{t+1}(i) y'_{l'_i}, & \text{其他} \end{cases}$$

其中:

$$\beta_t(s) = 0, \forall s > 2t$$

以上条件对应于图 17-4 中左图下部分路径不通的点。

CTC 前向和后向概率的计算都涉及多个概率 (范围为 $[0, 1]$) 的连乘, 这很容易导致出现数值计算问题, 常用的解决方案是对概率连乘取对数, 使乘法变加法, 对于极微小的概率, 取完对数后依然可能出现数值问题, 在实现上需要一些额外的技巧。

17.7 CTC 目标函数

前面已经介绍了 CTC 的基本原理和前/后向概率的递推计算方法。这一节我们将设定 CTC 的目标函数, 从而可以通过梯度下降的方法进行优化。

类似于普通的分类, CTC 的损失函数 O 定义为负的最大似然, 为了计算方便, 对似然取对数。

$$O = -\ln\left(\prod_{(x,z) \in S} p(z|x)\right) = -\sum_{(x,z) \in S} \ln p(z|x)$$

假设只看一个训练样本 (x, z) , 则对应的梯度为:

$$\frac{\partial O}{\partial y_k^t} = -\frac{\partial \ln p(z|x)}{\partial y_k^t} = -\frac{1}{p(z|x)} \frac{\partial p(z|x)}{\partial y_k^t}$$

由前向和后向的计算公式可知, 其乘积 $\alpha_t(s)\beta_t(s)$ 表示的是通过 \mathcal{B} 将输入 x 映射到 z (对应于之前的 l) 且 $\pi_t = z'_s$ 的所有路径 π 的概率加和:

$$\alpha_t(s)\beta_t(s) = \sum_{\pi \in \mathcal{B}^{-1}(z): \pi_t = z'_s} \prod_{t=1}^T y_{\pi_t}^t = \sum_{\pi \in \mathcal{B}^{-1}(z): \pi_t = z'_s} p(\pi|x)$$

这只是总体概率 $p(z|x)$ 的一部分, 因为这些路径 t 时刻必须经过 z'_s 。如果针对 t 时刻的所有可能求和, 我们得到:

$$p(z|x) = \sum_{s=1}^{|z'|} \alpha_t(s)\beta_t(s)$$

如果是对 y_k^t 求偏导 (注意, 这里 k 不是简单的下标, 而是代表非空字符或者 blank), 那么只有 t 时刻标注为 k 的路径需要考虑, 又因为是连乘, 所以偏导数为连乘项除以变量 y_k^t ;

其他项因为不含变量 y_k^t ，求偏导后为 0。公式表示如下：

$$\frac{\partial(\alpha_t(s)\beta_t(s))}{\partial y_k^t} = \begin{cases} \frac{\alpha_t(s)\beta_t(s)}{y_k^t}, & \text{如果 } k \text{ 出现在 } z' \text{ 中} \\ 0, & \text{否则} \end{cases}$$

对 $p(z|x)$ 积分得到：

$$\frac{\partial p(z|x)}{\partial y_k^t} = \frac{1}{y_k^t} \sum_{s \in \text{pset}(z,k)} \alpha_t(s)\beta_t(s)$$

其中 $\text{pset}(z,k)$ 表示的是 z' 中字符 k 出现的位置集合，即 $\text{pset}(z,k) = \{s : z'_s = k\}$ 。比如 $z = \text{banana}$ ，当 $k = a$ 时， $\text{pset}(z,k) = \{4, 8, 12\}$ ，因为 a 出现在 $z' = _b_a_n_a_n_a_$ 的这些位置。

代入原偏导公式后得：

$$\frac{\partial O}{\partial y_k^t} = -\frac{\partial \ln p(z|x)}{\partial y_k^t} = -\frac{1}{p(z|x)} \frac{\partial p(z|x)}{\partial y_k^t} = -\frac{1}{p(z|x)y_k^t} \sum_{s \in \text{pset}(z,k)} \alpha_t(s)\beta_t(s)$$

一般 y_k^t 为 Softmax 层输出：

$$y_k^t = \frac{e^{u_k^t}}{\sum_i e^{u_i^t}}$$

其中 u_k^t 为 Softmax 层输入，则与 Softmax 相关的偏导为：

$$\frac{\partial y_{k'}^t}{\partial u_k^t} = \begin{cases} \frac{e^{u_k^t} \sum_i e^{u_i^t} - e^{u_k^t} e^{u_k^t}}{(\sum_i e^{u_i^t})^2} = y_k^t - (y_k^t)^2, & \text{当 } k = k' \text{ 时} \\ \frac{0 - e^{u_{k'}}^t e^{u_k^t}}{(\sum_i e^{u_i^t})^2} = -y_k^t y_{k'}^t, & \text{其他} \end{cases}$$

合并则为：

$$\frac{\partial y_{k'}^t}{\partial u_k^t} = y_k^t * \delta(k = k') - y_k^t y_{k'}^t$$

其中 δ 为指示函数，即 $k = k'$ 成立时， $\delta(k = k') = 1$ ；否则 $\delta(k = k') = 0$ 。

$$\begin{aligned}
 \frac{\partial O}{\partial u_k^t} &= - \sum_i \frac{\partial O}{\partial y_i^t} \frac{\partial y_i^t}{\partial u_k^t} \\
 &= - \sum_i \left(\left(\frac{1}{p(z|x)y_i^t} \sum_{s \in \text{pset}(z,i)} \alpha_t(s)\beta_t(s) \right) (y_k^t * \delta(k=i) - y_i^t y_k^t) \right) \\
 &= - \frac{1}{p(z|x)} \sum_{s \in \text{pset}(z,k)} \alpha_t(s)\beta_t(s) + y_k^t \sum_i \left(\frac{1}{p(z|x)} \sum_{s \in \text{pset}(z,i)} \alpha_t(s)\beta_t(s) \right) \\
 &= y_k^t - \frac{1}{p(z|x)} \sum_{s \in \text{pset}(z,k)} \alpha_t(s)\beta_t(s)
 \end{aligned}$$

注意其中用到：

$$\sum_i \left(\frac{1}{p(z|x)} \sum_{s \in \text{pset}(z,i)} \alpha_t(s)\beta_t(s) \right) = \frac{1}{p(z|x)} \sum_i \left(\sum_{s \in \text{pset}(z,i)} \alpha_t(s)\beta_t(s) \right) = 1$$

17.8 CTC 解码基本原理

一旦 RNN 和 CTC 网络训练完毕，CTC 也存在很多路径，类似于隐马模型，解码（Decoding）过程就是要从众多路径中寻找最大概率的序列 l^* ，则：

$$l^* = \arg \max_l p(l|x)$$

17.8.1 最大概率路径解码

CTC 解码的第一种方法类似于传统的隐马模型，只是寻找最大概率路径 π^* ，用最大概率路径 π^* 的对应序列 $\mathcal{B}(\pi^*)$ 作为最大概率序列 l^* 的近似。即：

$$l^* \approx \mathcal{B}(\pi^*), \text{ 其中 } \pi^* = \arg \max_{\pi} p(\pi|x)$$

求最大概率路径的算法比较成熟，但是最大概率路径对应的序列往往不是最大概率序列。

如图 17-5 所示, 有两个时序的输出结果, 第一个时序对应于第一列的两个圆圈, 空心的为 blank, 概率为 0.8, 实心的为字母 V, 概率为 0.2; 第二个时序对应于第二列的两个圆圈, blank 概率为 0.6, 字母 V 概率为 0.4。

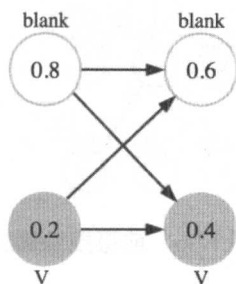


图 17-5 CTC 解码示例

图中最大概率路径为 $\pi_{1:2} = _ _$, 即两个 blank, 对应的序列为 $l = \text{blank}$, 相应概率为:

$$p(l = \text{blank}) = p(_ _) = 0.8 \times 0.6 = 0.48$$

而最大概率序列 $l^* = V$, 其对应概率为:

$$p(l^* = V) = P(VV) + P(_V) + P(V_) = 0.2 \times 0.4 + 0.8 \times 0.4 + 0.2 \times 0.6 = 0.52$$

可以看出, 在这个例子中最大概率路径和最大概率序列完全不同, 而这种不同在实际中也经常存在, 所以 CTC 解码通常不适合采用最大概率路径的方法。

17.8.2 前缀搜索解码

如图 17-6 所示, 前缀搜索解码是一种树形结构, 根节点为起始节点, 叶子节点为 e, 每个节点上的浮点数表示以根节点到该节点路径为前缀的所有路径的概率之和。叶子节点上的浮点数则是最终一个标注的本身概率。每一次迭代都选择最大概率的和目标串匹配的节点路径, 迭代结束的条件为: 以 e 结束的叶子节点概率大于其他所有合法前缀的剩余概率。在图 17-6 中, 目标串 l 为 XY, 以 e 结束的目标串 “_XYe” 概率为 0.3, 大于其他合法前缀的剩余概率, 比如 “_XX” 前缀对应的剩余概率为 0.1, “_XYY” 的剩余概率为 0.1。而其他前缀并不符合目标串为 XY 的要求。

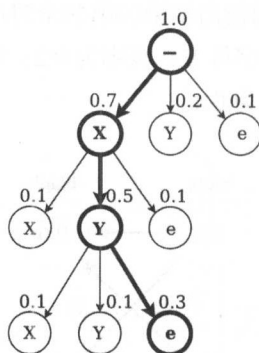


图 17-6 前缀搜索解码

如果不考虑性能，使用前缀搜索算法总能找到最大概率的候选序列。但是，这种多叉树的搜索空间随着树深度的增加呈指数级增长，在实际中需要结合一些其他的启发式方法。比如，考虑到 CTC 的输出序列夹杂很多 blank，如果某些 blank 的概率高于某个阈值，就以此 blank 为分隔点将输出序列分为多段，每一段单独进行前缀搜索解码。结合这种启发式方法，前缀搜索解码通常表现得比其他解码方式更优。但是这种方法还是会存在一些风险的，在实际中 RNN 之类模型输出的 blank 概率往往不太准确，从而导致切分和解码不准。

17.8.3 约束解码

一般 CTC 被用于手写和语音识别，最终的输出序列除考虑 RNN 之类模型本身的输出概率之外，还会结合语言模型等其他概率，这种同时考虑语言模型约束的 CTC 解码被称为约束解码。

原来普通的解码目标 $l^* = \arg \max_l p(l|x)$ 变为 $l^* = \arg \max_l p(l|x, G)$ ，假设给定 l 时，输入 x 和语言模型 G 条件独立，即：

$$p(x, G|l) = p(x|l)p(G|l)$$

则：

$$\begin{aligned}
 p(l|x, G) &= \frac{p(x, G|l)p(l)}{p(x, G)} \\
 &= \frac{p(x|l)p(G|l)p(l)}{p(x, G)} \\
 &= \frac{\frac{p(l|x)p(x)}{p(l)} \frac{p(l|G)p(G)}{p(l)} p(l)}{p(x|G)p(G)} \\
 &= \frac{p(l|x)p(l|G)p(x)}{p(x|G)p(l)}
 \end{aligned}$$

进一步，我们假设 x 和 G 条件独立，即：

$$p(x|G) = p(x)$$

则：

$$\begin{aligned}
 p(l|x, G) &= \frac{p(l|x)p(l|G)p(x)}{p(x|G)p(l)} \\
 &= \frac{p(l|x)p(l|G)}{p(l)}
 \end{aligned}$$

这种条件独立的假设在实际当中往往并不成立，比如语言模型和输入串往往与对应的语言存在依赖关系。但类似于朴素贝叶斯（Naive Bayesian）模型，这种条件独立的假设往往是非常合理的近似，尤其是在实际当中语言模型往往来自于独立的文本语料，而输入 x 则为手写的图片或者语音。

进一步，我们假设所有标注序列都是等概率的，则进一步推出：

$$\begin{aligned}
 p(l|x, G) &= \frac{p(l|x)p(l|G)}{p(l)} \\
 &\propto p(l|x)p(l|G)
 \end{aligned}$$

$$l^* \propto \arg \max_l p(l|x)p(l|G)$$

下面介绍一种约束解码方法——CTC 令牌传递算法（Token Passing Algorithm）^[9]，它可以根据一种简单的语言模型 G （比如二元语言模型）找到一个近似解。

假设语言模型 G 对应的词典 D 含有 w 个单词，则二元语言模型 $p(w|\hat{w})$ 表示的单词 \hat{w}

转换为 w 的概率, 注意这里 \hat{w} 和 w 都是词语 (字母序列), 可以看出:

$$p(w|\hat{w}) = \begin{cases} > 0, & \text{如果 } \hat{w} \text{ 是 } w \text{ 的真子串, 且 } w \in D, \hat{w} \in D \\ = 0, & \text{否则} \end{cases}$$

对任意的单词 w , 在其开头、结尾以及每两个字母之间都插入 blank, 得到的新单词记为 w' , 这与之前所讲的 l 和 l' 是一样的, 只是这里已精确到文本序列, 即单词。类似地, $|w'| = 2|w| + 1$ 。定义令牌 (Token) $\text{tok} = (\text{score}, \text{history})$, 其中 score 表示分数, history 表示已经访问过的单词。实际中, 令牌对应于输出序列中的某条特定路径, score 是对应路径的概率, history 是这条路径上经过的单词。

令牌传递算法的基本思想类似于维特比 (Viterbi) 算法, 利用动态规划不断迭代寻找每一步的最大得分令牌, 与语言模型相关的转换概率正好用于令牌之间的切换, 准确地说, 是从一个单词的最后一个状态转移到另一个单词的开始状态。

假设 t 为当前的时间步数, 输出序列的总时间步数为 T , s 为 w' 中对应片段的原串长度, 对应的最高得分令牌记为 $\text{tok}(w, s, t) = (\text{score}_{st}, \text{history}_{st})$, 该令牌对应的 score_{st} 为 t 时刻对应原串 $w_{1:s}$ 的路径中的最大概率, 而 history_{st} 则为沿途已经访问过的单词。此外, 定义输入令牌 $\text{tok}(w, 0, t)$ 为 t 时刻刚好到达单词 w , 输出令牌 $\text{tok}(w, -1, t)$ 为 t 时刻离开单词 w 的最高得分令牌。

CTC 令牌传递算法的伪代码^[8] 如下:

1. 初始化:
2. **for** $w \in D$ **do**
3. $\text{tok}(w, 1, 1) = \ln(y_b^1(w))$ // 此时, 令牌对应的 score 即时刻 1 为 blank 的概率
4. $\text{tok}(w, 2, 1) = \ln(y_{w_1}^1(w))$ // 此时, 令牌对应的 score 即时刻 1 为 w 第一个字母的概率
5. 如果 w 的长度为 1, 则:
6. $\text{tok}(w, -1, 1) = \text{tok}(w, 2, 1)$ // 此时, 单词 w 已结束, 因此为输出令牌
7. 其他:
8. $\text{tok}(w, -1, 1) = (-\infty, ())$ // 没有单词结束
9. 所有其他 s : $\text{tok}(w, s, 1) = (-\infty, ())$
10. 算法:
11. **for** $t = 2$ **to** T **do**

12. **if** 使用二元语言模型 (考虑前后两个的依赖关系):
13. 将所有输出令牌 $\text{tok}(w, -1, t - 1)$ 按照 score 增序排序
14. **else**:
15. 找到最高得分的输出令牌
16. **for** $w \in D$ **do**
17. **if** 使用二元语言模型:
18. $w^* = \arg \max_{\hat{w} \in D} [\text{tok}(\hat{w}, -1, t - 1).score + \ln p(w|\hat{w})]$ // w^* 到 w 的最大概
率单词 \hat{w}
19. $\text{tok}(w, 0, t) = \text{tok}(w^*, -1, t - 1)$ // 递推
20. $\text{tok}(w, 0, t).score += \ln p(w|\hat{w})$ // 多一个节点后的路径得分
21. **else**:
22. $\text{tok}(w, 0, t) =$ 最高得分的输出令牌
23. $\text{tok}(w, 0, t).history += w$ // history 集合中新增了单词 w
24. **for** segment $s = 1$ to $|w'|$ **do** // 利用递推公式迭代
25. $P = \{\text{tok}(w, s, t - 1), \text{tok}(w, s - 1, t - 1)\}$
26. **if** $w'_s \neq \text{blank} \ \&\& \ s > 2 \ \&\& \ w'_{s-2} \neq w'_s$ **then**
27. 将 $\text{tok}(w, s - 2, t - 1)$ 添加到 P 中
28. $\text{tok}(w, s, t) = P$ 中最高得分的令牌 // 递推公式
29. $\text{tok}(w, s, t).score += \ln y'_{w'_s}$ // 加上 t 时刻为 w'_s 的 log 概率
30. $\text{tok}(w, -1, t) = \text{tok}(w, |w'|, t)$ 与 $\text{tok}(w, |w'| - 1, t)$ 的最高得分
31. 终止:
32. $w^* = \arg \max_w \text{tok}(w, -1, T).score$ // 最大概率序列对应的单词
33. 输出: $\text{tok}(w^*, -1, T).history$

其中第 18~21 行为令牌传递递推公式, 类似于 CTC 的前向算法。

$$\text{tok}(w, s, t) = \beta_t(s) = \begin{cases} \arg \max_{i=(s-1) \rightarrow s} \text{tok}(w, i, t - 1), & \text{如果 } s \leq 2 \text{ 或者 } l'_s = \text{blank} \text{ 或者 } l'_{s+2} = l'_s \\ \arg \max_{i=(s-2) \rightarrow s} \text{tok}(w, i, t - 1), & \text{否则} \end{cases}$$

参考文献

- [1] M. W. Kadous. Temporal Classification: Extending the Classification Paradigm to Multivariate Time Series. PhD thesis, School of Computer Science & Engineering, University of New South Wales, 2002. URL <http://www.cse.unsw.edu.au/~waleed/phd/html/phd.html>.
- [2] H.A. Bourlard and N. Morgan. Connectionist Speech Recognition: A Hybrid Approach. Kluwer Academic Publishers, 1994.
- [3] A. Graves, S. Fernández, and J. Schmidhuber. Bidirectional LSTM networks for improved phoneme classification and recognition. In Proceedings of the 2005 International Conference on Artificial Neural Networks, Warsaw, Poland, 2005b.
- [4] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In Proceedings of the International Conference on Machine Learning, ICML 2006, Pittsburgh, USA, 2006.
- [5] S. Fernandez, A. Graves, and J. Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In Proceedings of the 2007 International Conference on Artificial Neural Networks, Porto, Portugal, September 2007a.
- [6] M. Liwicki, A. Graves, S. Fernández, H. Bunke, and J. Schmidhuber. A novel approach to online handwriting recognition based on bidirectional long short-term memory networks. In Proceedings of the 9th International Conference on Document Analysis and Recognition, ICDAR 2007, Curitiba, Brazil, September 2007.
- [7] A. Graves, S. Fernández, M. Liwicki, H. Bunke, and J. Schmidhuber. Unconstrained online handwriting recognition with recurrent neural networks. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, Advances in Neural Information Processing Systems 20. MIT Press, Cambridge, MA, 2008a.
- [8] A. Graves. Supervised Sequence Labelling with Recurrent Neural Networks. PhD thesis.
- [9] S. Young, N. Russell, and J. Thornton. Token passing: A simple conceptual model for connected speech recognition systems. Technical Report CUED/F-INFENG/TR38, Cambridge University Engineering Dept., Cambridge, UK, 1989.

第 4 部分

自然语言处理篇

自然语言处理简介

自然语言处理（Natural Language Processing, NLP）是人工智能和语言学相结合的交叉学科，主要研究如何让计算机处理并应用人类语言。可以说，计算机视觉和语音识别是人工智能领域的感知智能，而 NLP 属于人工智能领域的认知智能，因而相对也更难一些。在深度学习的发展过程中也与之类似，语音和图像提前获得突破，而 NLP 这两年才渐渐在机器翻译等领域大展身手。

NLP 至少包括两个方面。

- （1）语义理解：理解人类语言。
- （2）语言生成：机器生成人类可理解的语言。

而在这两个过程中，也需要涉及语言的解析，即识别语言结构、获取对应语义等。

18.1 NLP 的难点

NLP 相对较难，主要是因为语言有很多复杂的情况，比如歧义、省略、指代、重复、更正、倒序、反语等。

歧义产生的原因有多种，人类往往可以根据上下文来正确解析，而机器在这方面相比人类目前还稍逊一筹。

歧义至少有如下几种。

(1) 有些歧义是指代不明确带来的。比如“曾记否，我与你认识的时候，还是个十来岁的少年，纯真无瑕，充满幻想。”其中十来岁的少年指代不明，有可能指你，也有可能指我。

(2) 有些歧义是机器断句困难导致组合层次不同带来的。比如“我们四个人一组”可以理解为“我们/四个人一组”或“我们四个人/一组”；“这件事我办不好”可以理解为“这件事/我/办不好”或“这件事/我办/不好”。

(3) 有些歧义是结构关系不同导致的。比如“学生家长”可以理解为“学生的家长”或“学生和家長”；“出口食品”可以理解为动宾关系，也可以理解为偏正关系。

(4) 有些歧义是词语语义多带来的。比如“他想起来了”可以理解为“他想起床了”或者“他想起某件事情了”。

(5) 词类不同也可以带来歧义。比如“我要炒饭”中的“炒”可以是动词，也可以是形容词。

此外，很多新的品牌或网络用语也会带来歧义问题。

歧义的本质原因在于自然语言文本串与语义之间并非一对一的关系，一个相同或相似的文本串在不同的上下文中往往具有完全不同的语义。

18.2 NLP 的研究范围

NLP 在各个互联网公司都是不可或缺的，包括但不限于搜索、教育、医疗、金融、电商等领域公司。其主要研究范围如下。

(1) 分词：利用算法将一个汉字序列切分为一个个单独的词。比如将“我爱机器学习”切分为“我/爱/机器学习”。

(2) 词性标注：将分词结果中的每个单词标注为名词、动词、形容词、副词或其他词性的过程。

(3) 命名实体识别：识别文本串中具有特定物理意义的实体单词，比如人名、地名、机构名等。

(4) 关键词提取：提取文本串中若干个可以代表文章语义内容的词汇或短语。

(5) 自动摘要：也称为摘要提取，即根据文本语义内容提取较短的语句以概括语义。

(6) 主题模型：隐式主题模型如 Latent Semantic Analysis (LSA)、Probabilistic Latent Semantic Analysis (PLSA)、Latent Dirichlet Allocation (LDA) 等都是非常常见的研究领域。

(7) 依存句法分析：分析语言成分之间的依存关系，并揭示其语法树。

(8) 词嵌入 (Word Embedding)：将词采用向量表示。词嵌入从 2013 年左右开始就一直比较流行，可以说，词嵌入本身不是深度学习，但词嵌入是深度学习用于自然语言处理的基本前提。

(9) 机器翻译：利用计算机将一种自然语言转换成另一种自然语言的过程，两种自然语言分别称为源语言和目标语言。近年来，深度学习在机器翻译领域的应用也取得了较大的进展。

本书接下来将主要介绍深度学习在词性标注、依存句法分析、词嵌入、机器翻译四个方面的应用。

词性标注

词性标注 (Part-of-Speech Tagging/POS Tagging) 是自然语言处理领域的一个基础问题。词性标注是指根据句子上下文信息给句中单词指派正确的词性标签。英文词性标注有不同的标签集合定义, 本书中用的是 Penn Treebank 标签集合。词性标注的困难之处在于, 同一个英文单词在不同的句子中可能是不同的词性。例如: 1. I have a book. 2. I book a room., 在句子 1 中 book 的词性标签是名词 (NN), 而在句子 2 中 book 的词性标签是非第三人称单数动词 (VBP)。本章将主要介绍基于神经网络的词性标注模型。

19.1 传统词性标注模型

传统的词性标注方法有隐马尔可夫模型 (HMM) 和最大熵马尔可夫模型 (MEMM) 等。其中, HMM 是生成模型, MEMM 是判别模型。

基于 MEMM 的词性标注器抽取当前待标注单词附近的特征, 然后利用这些特征判别当前单词的词性。MEMM 是最大熵模型 (ME) 在处理序列模型方面的变种。其思想是在一串满足约束的标签中选出一个熵最大的标签。用 T 表示标签集合, t 表示其中的某一个标签, h 表示当前单词的上下文信息。这种模型可以用来估计句子 w_1, \dots, w_n 的标注 t_1, \dots, t_n 的概率, 公式^[1]如下:

$$p(t_1 \cdots t_n | w_1 \cdots w_n) = \prod_{i=1}^n p(t_i | t_1 \cdots t_{i-1}, w_1 \cdots w_n) \approx \prod_{i=1}^n p(t_i | h_i)$$

当前单词的上下文信息又叫作特征。根据在语料中出现的频次,可以将单词分为常见词和罕见词。常见词周围的特征包括:待标注的单词、待标注单词附近的单词、待标注单词附近已标注单词的词性标签等;罕见词的特征包括:单词的后缀、单词的前缀、单词是否包含数字、单词是否首字母大写等。

表 19-1 给出了特征值抽取的模板。定义函数 $f(t, h)$ 作为生成特征值的函数,根据第一行的模板可以生成如下特征:

$$f_1(t, h) = 1 \text{ iff } w_i = \text{make} \ \& \ t = \text{VB}$$

$$f_2(t, h) = 1 \text{ iff } w_i = \text{make} \ \& \ t = \text{NN}$$

表 19-1 特征模板^[2]

序号	特征类型	模板
1	通用型	$w_i = X \ \& \ t_i = T$
2	通用型	$t_{i-1} = T_1 \ \& \ t_i = T$
3	通用型	$t_{i-1} = T_1 \ \& \ t_{i-2} = T_2 \ \& \ t_i = T$
4	通用型	$w_{i+1} = X \ \& \ t_i = T$
5	稀疏型	$w_i \text{的后缀} = S \ \& \ S < 5 \ \& \ t_i = T$
6	稀疏型	$w_i \text{的前缀} = P \ \& \ 1 < P < 5 \ \& \ t_i = T$
7	稀疏型	$w_i \text{包含一个数字} \ \& \ t_i = T$
8	稀疏型	$w_i \text{包含一个大写字母} \ \& \ t_i = T$
9	稀疏型	$w_i \text{包含一个连字符} \ \& \ t_i = T$

模型可以使用最大似然的方法来训练,公式如下:

$$\begin{aligned} \hat{T} &= \arg \max_T P(T|W) = \arg \max_T P(t_1 \cdots t_n | w_1 \cdots w_n) \approx \arg \max_T \prod_{i=1}^n P(t_i | h_i) \\ &= \arg \max_T \prod_{i=1}^n \frac{\prod_{j=1 \cdots K} \exp(w_j f_j(h_i, t_i))}{\sum_{t'_i \in T} \prod_{j=1 \cdots K} \exp(w_j f_j(h_i, t'_i))} \end{aligned}$$

HMM 模型与 MEMM 模型的概率表示和求解都不相同。基于 HMM 的词性标注模型的目标函数如下。HMM 相关的资料很多,这里就不赘述了。

$$\hat{T} = \arg \max_T P(T|W) = \arg \max_T P(t_1 \cdots t_n | w_1 \cdots w_n) \approx \arg \max_T \prod_{i=1}^n \overbrace{P(w_i | t_i)}^{\text{发射概率}} \overbrace{P(t_i | t_{i-1})}^{\text{转移概率}}$$

HMM 和 MEMM 存在同一个问题,就是只能从一个方向预测接下来的标注。但是在很多情况下当前单词后面的标注信息对当前标注的反馈也非常重要。要解决这个问题有很多种方法。一种方法是采用更强的模型,比如条件随机场(CRF)。但是条件随机场的计算开销太大,并且对标注效果的提升有限。还有一种方法是斯坦福的词性标注器^[3]中用到的,一种叫作循环依赖网络(Cyclic Dependency Network)的模型,这种模型本质上是 MEMM 的变种。

19.2 基于神经网络的词性标注模型

传统的词性标注需要从句子中抽取一系列精心设计的特征。Bengio 将词向量引入自然语言处理中,大大减少了特征工程的工作量。

首先介绍一个简单的基于神经网络模型的词性标注器^[2]。模型从左向右依次标注句子中的单词,对于当前单词,抽取周围一定窗口大小内的特征,然后将其作为特征向量送入前馈神经网络分类器。这个分类器会预测出当前单词在上下文中最可能的词性。

如图 19-1^[2]所示,整个神经网络分为多层。第一层把每个单词映射到一个特征向量,得到单词级别的特征;第二层利用滑动窗口得到单词上下文的特征向量,不像传统的词袋方法,这个方法保留了窗口内单词的顺序关系。同时也可以加入其他特征,如单词是否首字母大写、单词的词干等。单词上下文的特征向量被送入后续的隐层,最终到达输出层。对于每个单词,输出层会有 n 个输出,其中 n 是词性标签集合的大小。每个输出值都可以看作该词性标签的得分,其中最大的输出值对应的标签就是当前单词最可能的词性。

在计算上下文特征时只考虑当前单词附近窗口大小为 k 范围内的单词,这种方法叫作窗口方法(Window Approach)。将整个句子的单词特征向量送入后续网络中,这种方法叫作句子方法(Sentence Approach)。对于词性标注来说,句子方法并不能带来明显的效果提升,但是对于自然语言里的某些任务,如语义角色标注(SRL),句子方法带来的效果提升会比较明显。此外,因为句子长度一般是不定的,所以在使用句子方法的神经网络模型中会增加卷积层。

一般来说,神经网络模型的优化目标是最大化对数似然函数。公式右侧的 $\sum_{(x,y) \in T} \log p(y|x, \theta)$ 是模型的对数似然函数,模型的优化目标就是最大化这个函数,得到参数估计 θ 。其中 T 是训练数据集, x 是输入的特征向量, y 是对应的标注, θ 是模型的参数, $p(\cdot)$ 是神经网络输出层的输出结果。

$$\theta \mapsto \sum_{(x,y) \in T} \log p(y|x, \theta)$$

单词级别的最大似然是通过每个单词分别计算得到的。给定一个输入特征向量 x , 设模

型的参数为 θ ，输入 x 的标注的索引为 y 。 $f_{\theta}^j(x)$ 表示的是模型在给定 x 和 θ 的情况下，第 j 个词性得到的分数。根据 Softmax 操作，对于一条训练数据 (x, y) 来说，对数似然函数如下：

$$\log p(y|x, \theta) = \log \frac{e^{f_{\theta}^y(x)}}{\sum_j e^{f_{\theta}^j(x)}} = f_{\theta}^y(x) - \log\left(\sum_j e^{f_{\theta}^j(x)}\right)$$

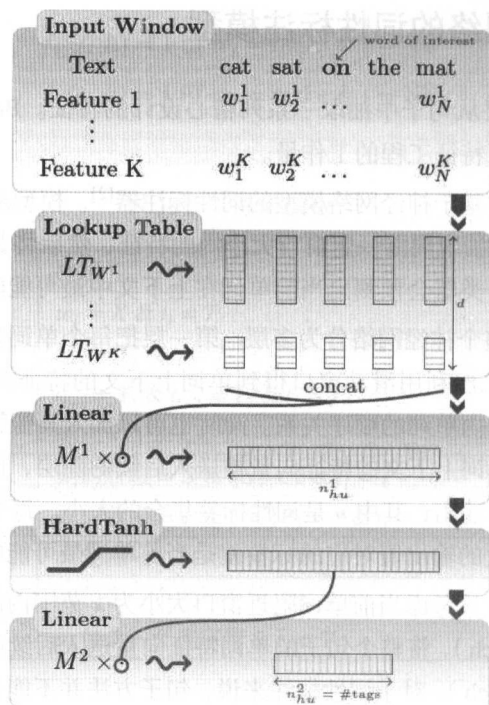


图 19-1 神经网络模型（窗口方法）^[2]

对于自然语言里的某些任务，如语义角色标注（SRL），需要用到句子级别的对数似然函数。不过，对词性标注效果提升不大，这里就不详细介绍了。

上面使用的是有监督的训练方法。此外，还可以借助大量的无标注数据来训练语言模型，从而得到更好的单词特征向量表示。用无监督训练得到的词向量初始化词性标注模型的词向量，能明显提升词性标注的准确率。值得一提的是，越大的语料训练出的模型越好。不过，这个训练过程比较缓慢，可以通过一点点地增加语料迭代优化。

19.3 基于 Bi-LSTM 的神经网络词性标注模型

在神经网络中常见的词向量模型是给词汇表建立一个查找表 (Lookup Table)，每个单词都可以在查找表中找到一个对应的词向量 (如图 19-2 所示)，然后多个词向量再组合成后续神经网络层的输入。

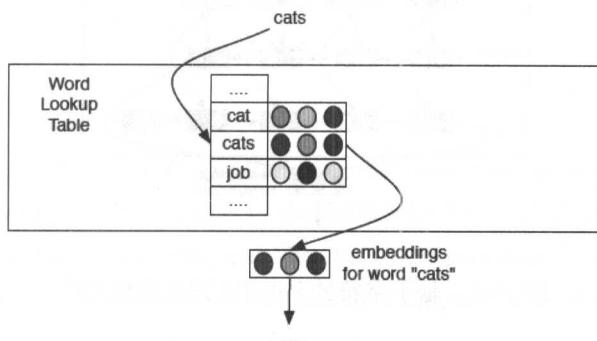


图 19-2 词向量查找表^[4]

普通的词向量结合大量语料可以学习到单词间语义 (Semantic) 和语法 (Syntactic) 上的相似性。举个例子，模型可以学到 *cats*、*kings*、*queens* 之间的线性相关性与 *cat*、*king*、*queen* 之间的线性相关性一样。不过模型并不能学到前面这组单词是由后面这组单词在末尾加 *s* 得到的。也就是说，对于没见过的单词，普通的词向量模型是没有办法的，即使这些单词是词汇表中单词的变形或者组合。除此之外，英语的词汇表是非常庞大的，想要建立一个完善的查找表并不是一件容易的事情。

普通的词向量模型查找表过于庞大，于是就有人提出将单词拆成更小的单元——词素 (Morpheme)。不过，词素的切分本身又要依赖于词素解析器 (Morphological Analyzer)。因此又有人提出了基于字符的词向量 (Character-Based Embedding of Words) 模型，在 Ling 等人 2015 年发表的论文中也叫 C2W (Compositional Character to Word) 模型^[4]。

基于字符的词向量模型的输入、输出和普通的词向量模型是一样的，因此在神经网络模型中这两种模型可以相互替换。与普通的词向量模型类似，基于字符的词向量模型是给字符集合建立一个查找表。字符集合包括大小写字母、数字、标点等。每个字符都可以在查找表找到对应的字符向量。每个单词都可以看成一串字符，将单词中的字符串对应的词向量从左到右依次送入 LSTM 模型，再从右向左依次送入 LSTM 模型。两个方向的 LSTM 模型生成的结果组合生成当前单词的词向量，这样就可以利用 Bi-LSTM 模型得到单词的向量表示。整个过程如图 19-3 所示。

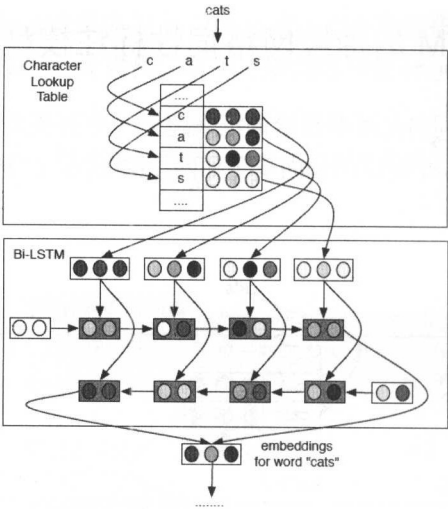


图 19-3 基于字符的 Bi-LSTM 词向量模型^[4]

有了词向量模型就可以构建词性标注模型了。下面介绍用来做词性标注的 Bi-LSTM 模型架构，如图 19-4 所示。Bi-LSTM 的输入是一串单词的特征值，这些特征值既可以用普通的词向量模型得到，也可以用基于字符的词向量模型得到。按照单词从左向右的顺序，把这些单词对应的词向量送入 LSTM 模型；同理，按照单词从右向左的顺序，把这些单词的词向量送入 LSTM 模型。两个 LSTM 模型的输出线性组合成 Bi-LSTM 的输出，将这个输出送入 Softmax 层，得到最终的词性标注结果。

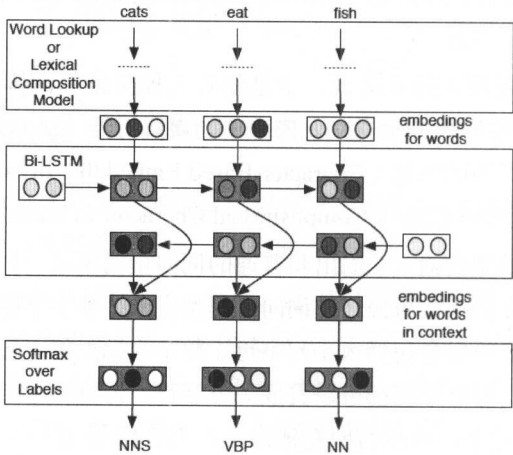


图 19-4 词性标注的 Bi-LSTM 模型架构^[4]

相对于普通的词向量模型，基于字符的词向量模型减少了很多参数。不过，因为英文中单词构成的复杂性，该模型在词性标注上的表现并没有超越现有模型。因为，虽然基于字符的词向量模型可以学习 *ed, ily* 这种形变特征，但是英文中有些字符构成很像的单词之间的差异却很大。比如 *lesson* 和 *lessen*，虽然从字符角度看起来很像，但是它们的含义却完全不同。不过，在一些单词形态更丰富的语言（如：土耳其语）中，基于字符的词向量模型的表现优于普通模型。

参考文献

[1] Toutanova, K., & Manning, C. D.. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics, 13, 63-70. 2000.

[2] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P.. Natural Language Processing (Almost) from Scratch. Journal of Machine Learning Research, 12, 2493-2537. 2011.

[3] Toutanova, K., Klein, D., & Manning, C. D.. Feature-rich part-of-speech tagging with a cyclic dependency network. In Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1 (NAACL'03), 252-259. 2003.

[4] Ling, W., Luis, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Trancoso, I.. Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation. Emnlp-2015, 1520-1530.

依存句法分析

本章主要介绍神经网络在 NLP 中的一个应用——SyntaxNet^[1]。

SyntaxNet 是 Google 发布的一款基于 TensorFlow 的 NLP 框架。随之一同发布的，是 Google 在 SyntaxNet 框架下开发设计的 NLP 程序 Parsey McParseface，主要用于词性标注和依存句法分析。其中依存句法分析取得了优异的成绩。

下文将不再区分 SyntaxNet、Parsey McParseface 以及 Parsey McParseface 的依存句法分析功能等概念，统一以 SyntaxNet 命名。同时，统一使用了一长一短两个例句：

“There’s a possibility of a surprise” in the trade report, said Michael Englund, director of research at MMS.

He has good control.

长句来自于 CoNLL 2009 的样例数据^[2]，短句来自于一篇 SyntaxNet 的相关论文^[3]。

SyntaxNet 的依存句法分析具有如下特性。

- 程序没有使用 RNN (LSTM)。
- 程序取得了超过 RNN 的准确率。

NLP 处理的对象主要是句子、文章，可以将这些数据看作单词或者字母的序列。序列长度不定，相关窗口较大。例如上文的短例句的序列长度只有 5，而长例句的序列长度却有 24。长例句中 possibility 和 report 有相关关系，但是却不相邻，跨度有 8 个长度。

普通神经网络处理 NLP 序列，输入数据一般是定长的向量，对应 NLP 的定长窗口。这

与 NLP 序列的变长、长相关特点相矛盾。RNN 处理这些问题有先天的优势，在很多应用中都取得了不俗的成绩。但 RNN 也有缺点：训练难度大，性能低下。

SyntaxNet 没有使用 RNN，保证了性能，但是却取得了等同甚至超过 RNN 的准确度。据开发者介绍，SyntaxNet 的准确度为 94.15%，平均每个单词的处理时间不到 2ms。回避 RNN 并且超越 RNN，正是 SyntaxNet 的价值所在。

考虑到读者可能对 NLP 相关技术比较陌生，本章首先介绍依存分析的背景知识。同时，SyntaxNet 能够成功，在于它应用了一系列技术，将 NLP 问题转化为可操作的机器学习问题，本章将对这些技术进行介绍。

20.1 背景

用一句话来说，依存句法分析就是对句子进行句法分析，生成一个句法依赖树的过程。

依存句法分析会有一些前置处理，如图 20-1 所示，首先对句子进行 Token 化，然后对 Token 序列进行词性标注。在这之后，才对标注后的 Token 序列进行分析。Token 化可以使用纯规则或者简单的模式匹配；词性标注任务相对简单，有自己成熟的解决方案。SyntaxNet 也实现了词性标注，其技术细节与依存句法分析类似，却更加简单，这里不再进行介绍。

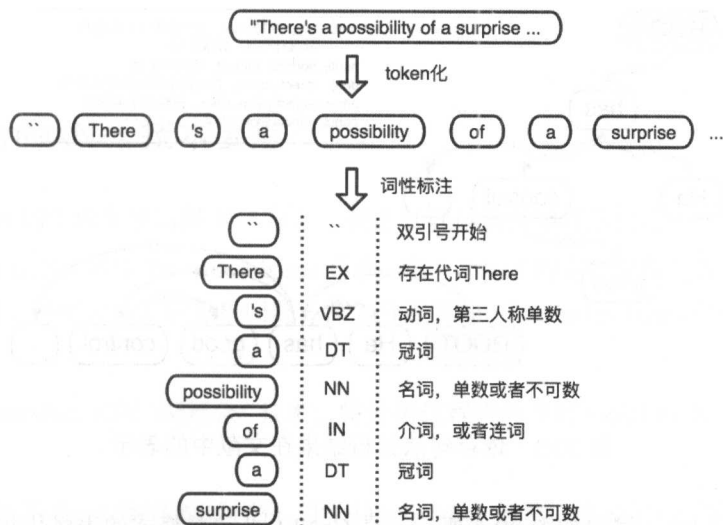


图 20-1 Token 化和词性标注

常见的分析树有两种——依存句法分析树和短语结构分析树，分别简称为依存树和短语树，如图 20-2 所示。

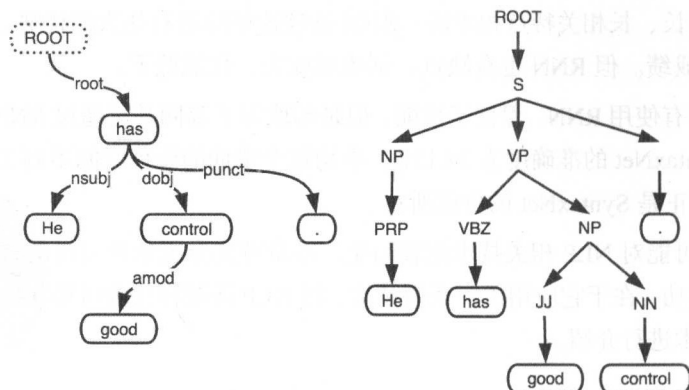


图 20-2 依存树及短语树

这两种树都可以反映句子的结构和内在关系，但也有所区别。SyntaxNet 只能生成依存句法分析树。

“树”只是一个概念上的表示，在各种文献中，会表示成“被拍扁在原文上”的树，如图 20-3 所示。附有标签的带箭头弧线用来表示依存关系：标签表示依存类型，弧线表示从属。

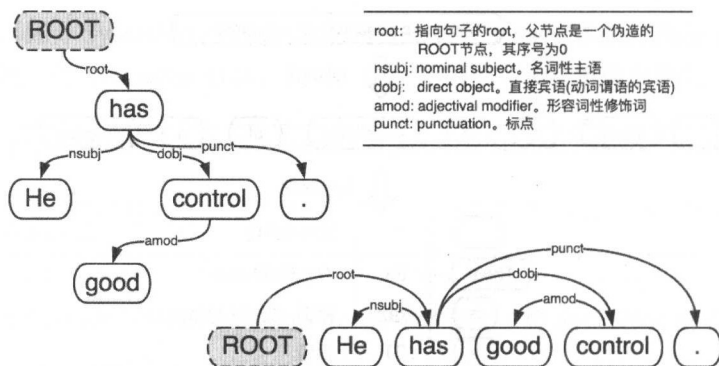


图 20-3 依存句法分析结果在文献中的表示

SyntaxNet 的输入输出如图 20-4 所示，是 CoNLL^[2] 语料格式的表格化数据。一个句子构成一张表，每行代表一个 token，其中每列表示该 token 的各种属性。对于依存句法分析而言，读取表格中的文本列（FORM）和词性标注列（POSTAG），输出标签列（DEPREL）和位置列（HEAD）。

ID	FORM	LEMMA	POSTAG	HEAD	DEPREL
1	``	``	``	3	P
2	There	there	EX	3	SBJ
3	's	be	VBZ	15	OBJ
4	a	a	DT	5	NMOD
5	possibility	possibility	NN	3	PRD
6	of	of	IN	5	NMOD
7	a	a	DT	8	NMOD
8	surprise	surprise	NN	6	PMOD
9	''	''	''	8	P
10	in	in	IN	8	LOC
11	the	the	DT	13	NMOD
12	trade	trade	NN	13	NMOD
13	report	report	NN	10	PMOD
14	,	,	,	15	P
15	said	say	VBD	0	ROOT
16	Michael	michael	NNP	17	NAME
17	Englund	englund	NNP	15	SBJ
18	,	,	,	17	P
19	director	director	NN	17	APPO
20	of	of	IN	19	NMOD
21	research	research	NN	20	PMOD
22	at	at	IN	19	NMOD
23	MMS	mm	NNS	22	PMOD
24	.	.	.	15	P

图 20-4 CoNLL 语料

20.2 SyntaxNet 技术要点

SyntaxNet 的处理框架如图 20-5 所示。整个处理过程是循环迭代的。

(1) SyntaxNet 利用 Transition-based 系统, 将一个句子的完整分析过程转换为一系列的子过程; 同时, 还将每一步的分析问题转换为分类问题。Transition-based 系统还维护了句子的状态。

(2) SyntaxNet 利用“模板化”技术, 将分析过程中每个时刻的句子状态编码为定长的 one-hot 向量。

(3) 分类器是一个深度神经网络。利用 Embedding 技术, 将表示句子状态的 one-hot 向量拼接为定长输入。分类器根据输入信息, 得到分析结果 (Transition-based 系统的操作), 输出一组概率值。

(4) 最后, SyntaxNet 还使用了 Beam Search 技术, 防止分析结果陷入局部次优解。

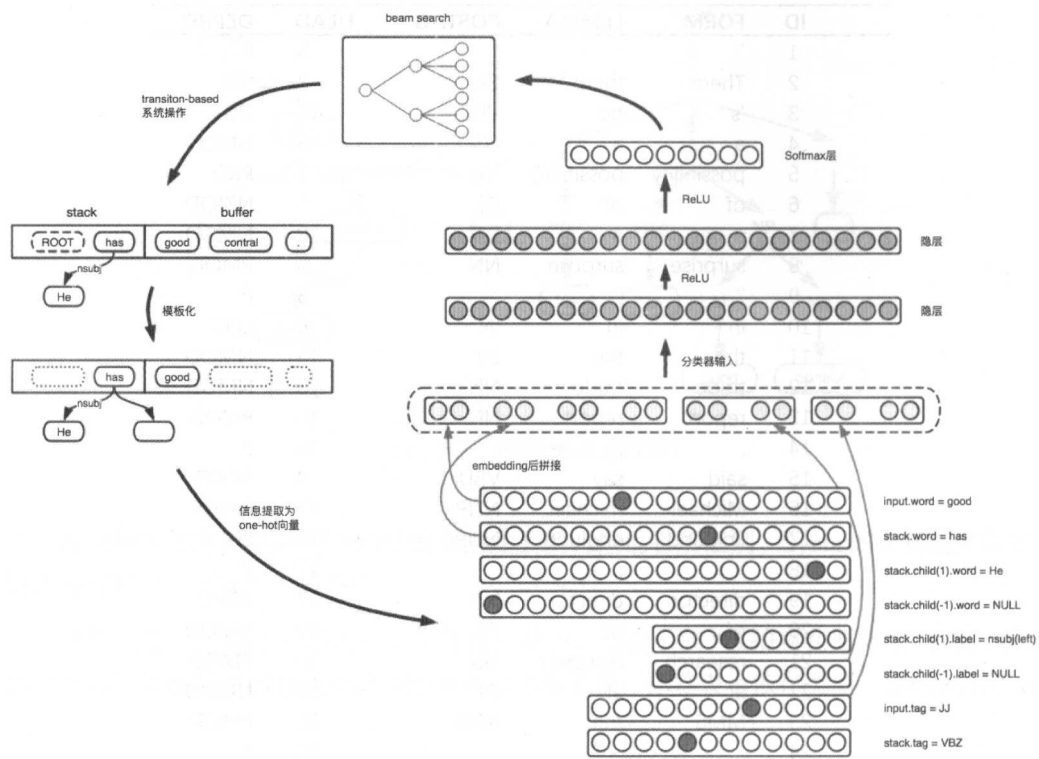


图 20-5 SyntaxNet 的处理框架

SyntaxNet 和其他机器学习/神经网络任务有很多共同的技术点，这里就不再介绍了。下面依次介绍 SyntaxNet 中三个特有的技术要点。

20.2.1 Transition-based 系统

如果使用端到端的风格（输入就是待分析的句子，而输出是一个标签序列，对应一整棵语法树），方案可能无法落地：不仅需要模型有“超凡的”NLP 能力，而且分析结果的变化形式也太多了（分类数是无穷大的）。

SyntaxNet 利用 Transition-based 系统，将任务拆解成一系列可重复的、相对简单的分类任务，从而使方案落地。

Transition-based 系统以 token 为粒度，用以下三部分表示句子状态。

(1) stack。保存着正在处理的 token。在初始情况和终止情况下，stack 中只有一个虚

构的 token——ROOT。SHIFT 操作将 token 从 buffer 传递到 stack 顶部。通过 LEFT-ARC、RIGHT-ARC 操作，将顶部的 token（top2 之一）从 stack 中移出，然后拼接到树枝上。

（2）buffer。保存着尚未处理的 token。在初始情况下保存着句子的所有 token。如果发生 SHIFT 操作，则将头部的 token 传递到 stack 中。在终止情况下 buffer 为空。

（3）树枝区。保存着已经处理过的 token。在树枝区中是一棵棵子树，其树根是 stack 中的某个 token。在初始状态下，没有任何子树。在终止条件下，生成一颗以 ROOT 为根的语法树。

Transition-based 系统同时还定义了三类操作。

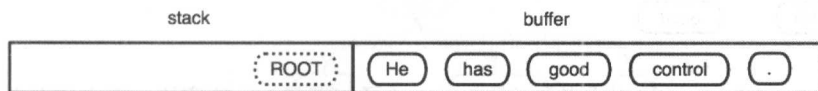
（1）SHIFT。操作的 token 总是位于 buffer 的头部（最左边），将其从 buffer 的头部移动到 stack 顶部（最右边）。当 stack 中的信息不足时进行此操作。

（2）LEFT-ARC。在 stack 顶部（最右边）的两个 token 之间创建一条向左的弧线（从顶部第一个到顶部第二个），然后将左侧的（顶部第二个）token 移出 stack。当发现 stack 顶部的两个 token 有依存关系时进行此操作。

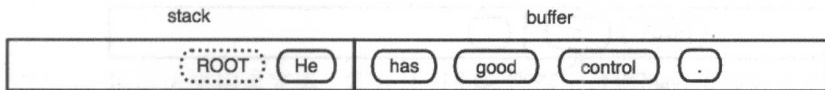
（3）RIGHT-ARC。类似于 LEFT-ARC，不过所创建的弧线的方向不同——创建一条向右的弧线，并将右侧的 token 移出 stack。

只是进行文字说明可能会让读者难以理解，下面举例说明 Transition-based 系统生成语法树的过程。

（1）初始条件。此时句子已经 Token 化并且完成了前置处理（包括词性标注）。所有 token 都在 buffer 中，stack 中只有一个虚构的 token。



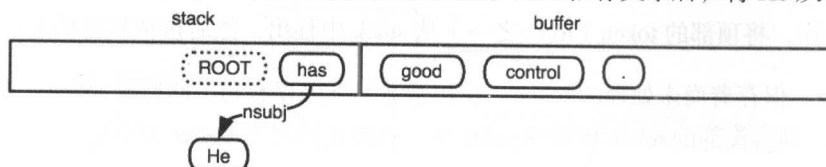
（2）SHIFT。将 He 移动到 stack 顶部。



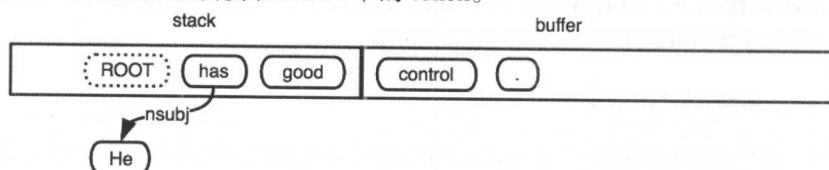
（3）SHIFT。将 has 移动到 stack 顶部。



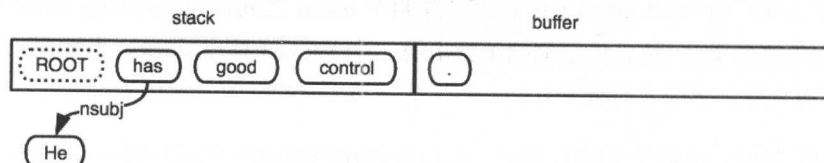
- (4) LEFT-ARC。He 是 has 的名词性主语，建立依存关系后，将 He 从 stack 中移出。



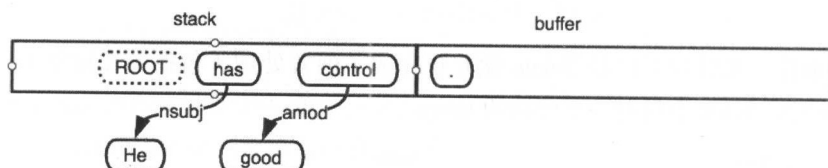
- (5) SHIFT。继续读取 buffer 中的 token。



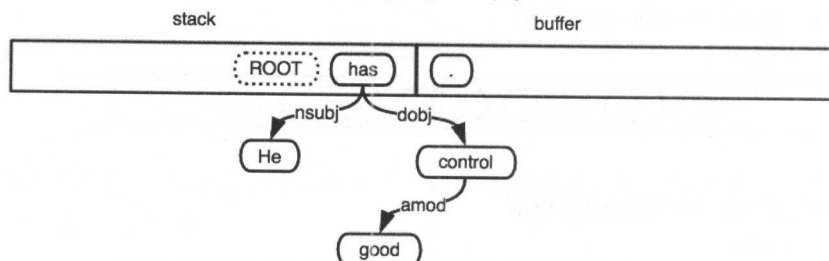
- (6) SHIFT。继续读取 buffer 中的 token。



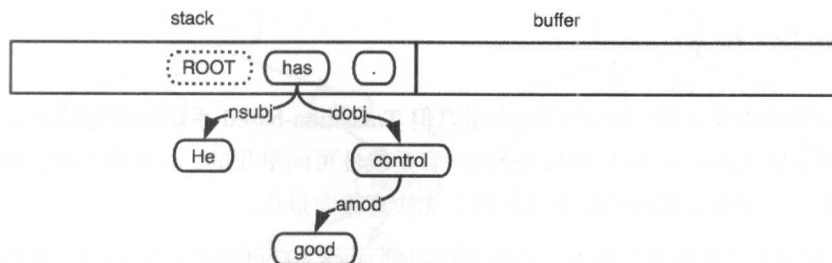
- (7) LEFT-ARC。good 是 control 的形容词性修饰词，建立依存关系后，将 good 从 stack 中移出。



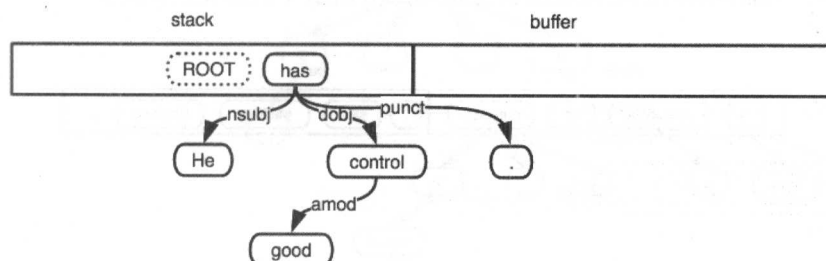
- (8) RIGHT-ARC。control (或者说 good control 子句) 是 has 的宾语，建立依存关系后，将 control 及其对应的树枝部分一起移动到 has 下。



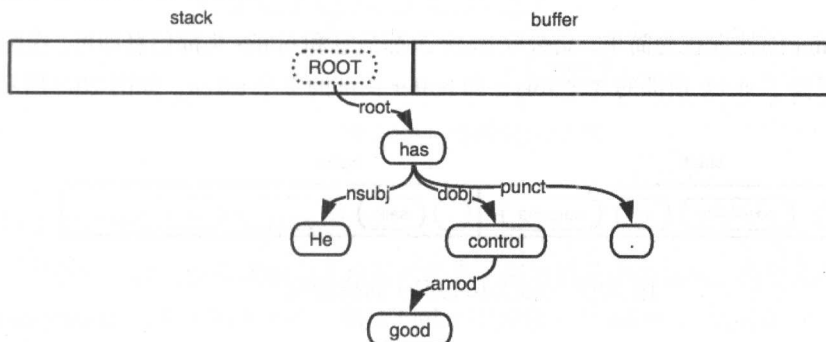
- (9) SHIFT。将最后一个 token 移动到 stack 中。buffer 为空后，不会再发生 SHIFT 操作。



(10) RIGHT-ARC. has 和句末标点适用于 punct 依存关系。



(11) 最后，stack 中只剩 ROOT 和 has 两个 token。使用 RIGHT-ARC 操作构建 root 依存关系后，将 has 移出 stack。



至此，一棵语法树构建完毕。需要注意的是，整个过程的总步骤数是由 token 数决定的——如果一个句子有 N 个 token，则会发生 N 次 SHIFT 操作和 N 次 LEFT-ARC/RIGHT-ARC 操作，总步骤数是 $2N$ 。

SyntaxNet 实现了一个多层神经网络分类器。根据当前 Transition-based 系统的状态，在 SHIFT、LEFT-ARC、RIGHT-ARC 三类操作中做出正确选择。

20.2.2 “模板化”技术

神经网络分类器的输入是一个定长的向量。但 Transition-based 系统的状态信息并不是定长的，也不是向量化的——stack 和 buffer 的内容是线性可向量化的，但长度不定；树枝部分不仅规模不定，还要考虑树枝的标签及形状，无法直接量化。

SyntaxNet 使用了“模板化”技术：不论当前时刻 stack 和 buffer 的内容如何，树枝区多么复杂，都可以通过模板化将系统的状态转换为定长的向量，作为分类器的输入。直接解释模板化较为困难，下面通过例子对模板化进行解释。

在某一时刻，Transition-based 系统的状态如图 20-6 所示（依存关系被隐去）。

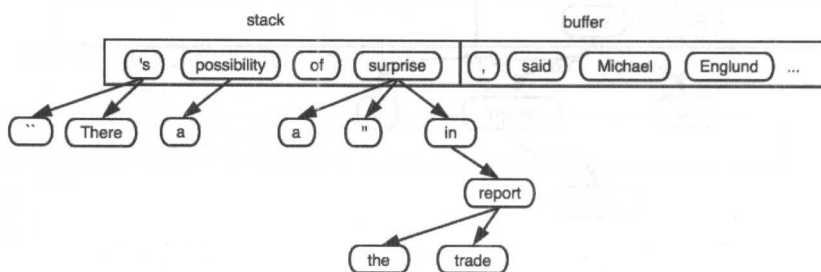


图 20-6 某一时刻 Transition-based 系统的状态

stack 和 buffer 的处理较为简单，只截取 stack 顶部信息和 buffer 头部信息即可。在图 20-6 所示的例子中保留了 stack 顶部的 3 个 token 和 buffer 头部的 2 个 token，如图 20-7 所示。

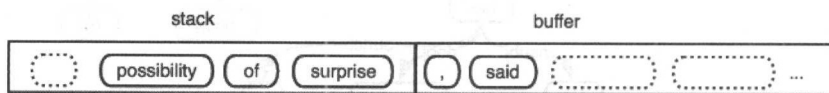


图 20-7 stack 和 buffer 的模板化

树枝部分的处理较为复杂，首先选择栈顶固定数目的树枝，离栈顶太远的树枝直接忽略，然后将树枝填写到“树的模板”中。例如，模板规定，栈顶的第一个树枝只考虑 6 个节点，第一层 4 个：最左、次左、最右、次右；第二层 2 个：最左-最左、最右-最右。不存在的节点置为 NULL。其过程和结果如图 20-8 所示。

模板化的结果如图 20-9 所示。在整个结果中信息的数目是确定的（12 个），这样就可以转换为定长向量了。例子中的“模板”较小，实际上 SyntaxNet 共抽取了 52 个信息：20 个 word、20 个 tag 和 12 个 label。

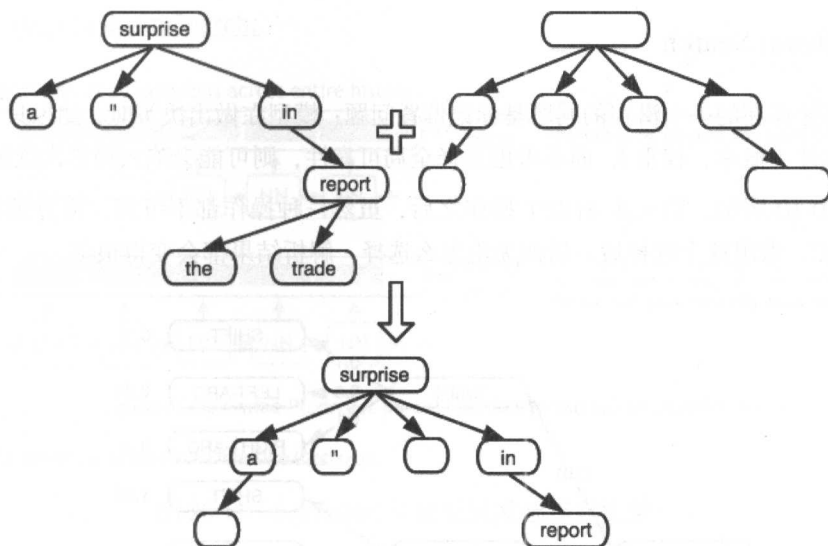


图 20-8 单个树枝的模板化

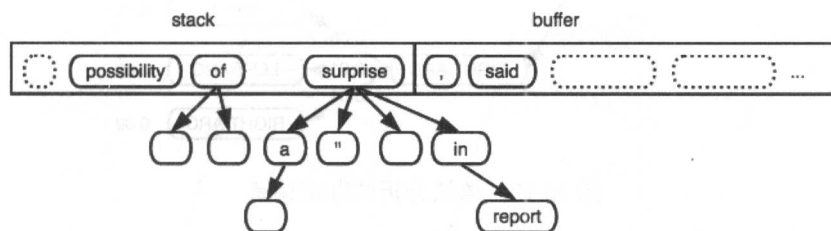


图 20-9 模板化的结果

模板化的作用类似于 LSTM 中的门：

一是忽略了和当前步骤相关度较低的信息。例如例子中的情况，模型此刻只是需要判断 of 和 surprise 的关系（如果无关，则从 buffer 中读取一个 token）。此时离 of 和 surprise 较远的单词，如很远处的 director，还有层数很深的树枝，如修饰 report 的 the trade，对判断的影响微乎其微。这些低相关的信息都通过模板化过滤掉了。

二是保留了长跨度高相关的信息。例如 possibility，因为句子的细枝末节部分已经放到树枝部分，所以离 surprise 更近了。还有 in report，因为处于栈顶 token 的树枝中，也被模板考虑在内。

20.2.3 Beam Search

词性标注存在的一个潜在的问题是标记偏置问题：模型在做出决策时，如果只考虑局部选择的可靠性（概率、权重），而不考虑选择全局可靠性，则可能会陷入局部次优解。

如图 20-10 所示，第一步 SHIFT 操作之后，虽然三种操作都不可靠，但分类器仍选择 RIGHT-ARC。做出这个选择后，后面无论怎么选择，解析结果都会变得很差。

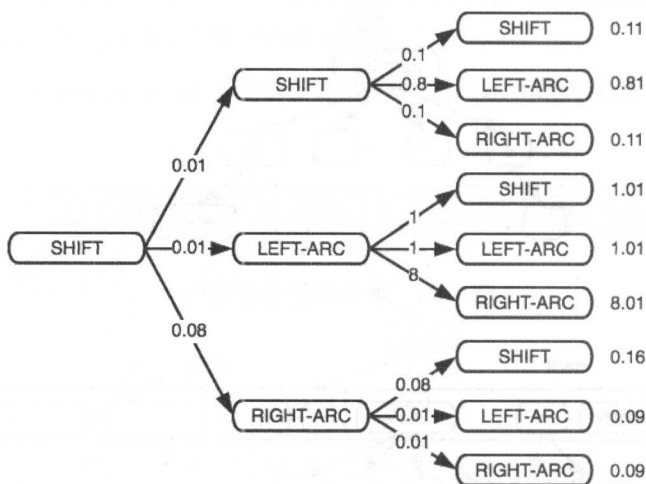


图 20-10 语法分析的标记偏置

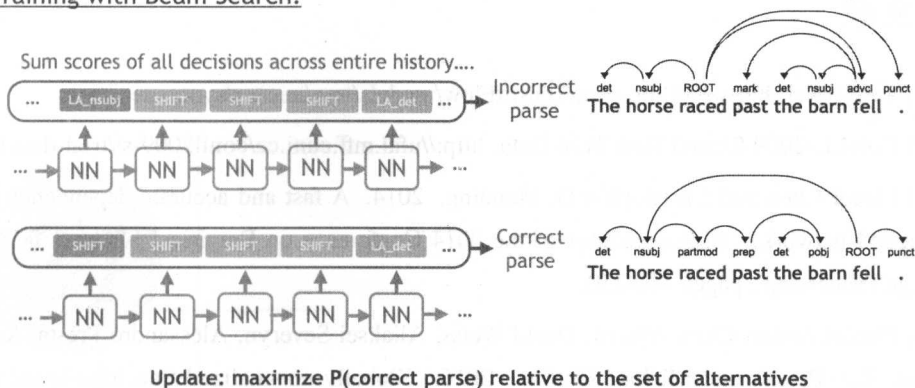
导致出现这种情况的句子有一个专门的名字：garden path sentence。SyntaxNet 使用 Beam Search 技术来解决 garden path sentence 的问题，如图 20-11 所示。

Beam Search 可以认为是保留次优解的广度优先搜索。普通的广度优先搜索保留所有的历史路径，而 Beam Search 只保留 TOP-N（称为 Beam Size）的历史路径。一个 Beam Size 为 2 的 Beam Search 的过程如图 20-12 所示。

Beam Search 无法彻底规避局部次优解。如图 20-10 所示，当 Beam Size 为 2 时（同样权重时位置偏上的优先），仍然无法取得最优解。如果 Beam Size 过大，又会影响效率。在实际使用中要在性能和效果上进行一定的折中。

在 Beam Search 获取路径权重上，SyntaxNet 也使用了一些小技巧：不是简单地使用 Softmax 层的输入或者输出，而是构建了一个小神经网络，使用大神经网络的隐层信息作为输入，输出一个权重值供 Beam Search 参考^[4]。

Training with Beam Search:



Globally Normalized SyntaxNet Architecture (Overview)

图 20-11 SyntaxNet 从全局出发选择最优解^[1]

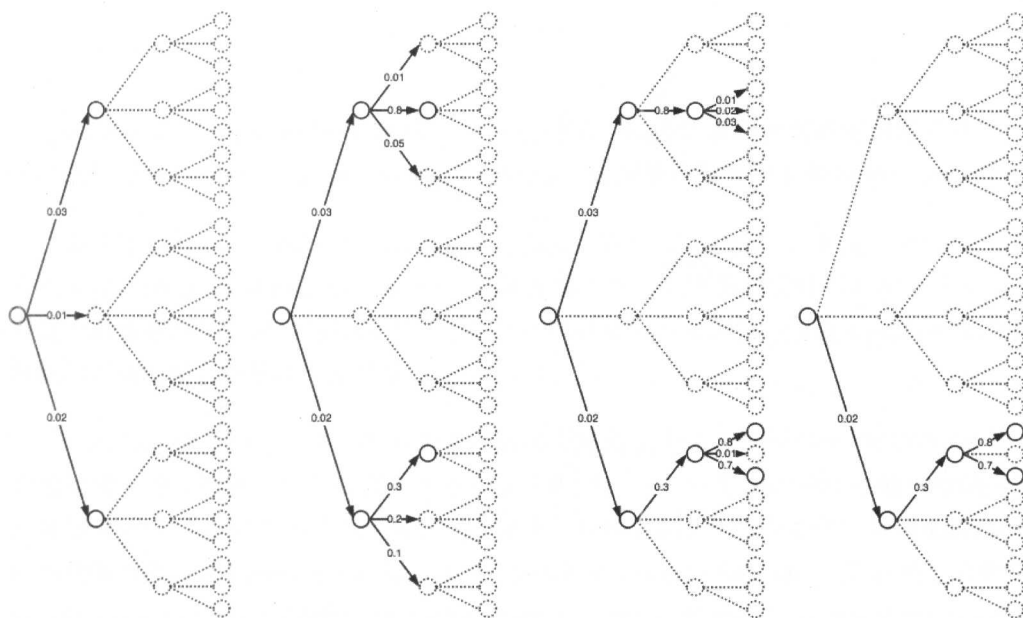


图 20-12 Beam Search 过程

至此，SyntaxNet 的三个主要技术要点全部介绍完毕。实际上，SyntaxNet 还做了很多细节的优化，例如自动生成语料，对 Beam Search 进行性能优化等。由于篇幅所限，这里就不一一介绍了，感兴趣的读者可以从 SyntaxNet 项目主页获得更多的资料^[1]。

参考文献

- [1] SyntaxNet. <https://github.com/TensorFlow/models/tree/master/syntaxnet>.
- [2] CoNLL-2009 Shared Task Trial Data. <http://ufal.mff.cuni.cz/conll2009-st/trial-data.html>.
- [3] Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing , pages 740-750.
- [4] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. Available at arXiv.org as arXiv:1603.06042, March.

21

word2vec

word2vec^{[1][2]} 是 Google 在 2013 年年中开源的一款将词表征为实数值向量的高效工具，采用的模型有 CBOW（Continuous Bag-Of-Words，连续的词袋模型）和 Skip-Gram 两种。

需要注意的是，word2vec 不是深度学习模型，它只有一个隐层。但是 word2vec 带来的分布表示（Distributed Representation），也就是每个单词或者其他实体都对应一个向量，而且这个向量在高维空间中的位置还蕴含着与单词语义相关的信息，使得 word2vec 成为很多深度学习问题研究的基础，尤其是在 NLP 领域。

word2vec 通过训练，可以把对文本内容的处理简化为 k 维向量空间中的向量运算，而向量空间中的相似度可以用来表示文本语义上的相似度。因此，word2vec 输出的词向量可以被用来做很多与 NLP 相关的工作，比如聚类、找同义词、词性分析等。而 word2vec 被人广为传颂的地方是其向量的加法组合运算（Additive Compositionality），其官网上的例子是： $\text{vector}(\text{'Paris'}) - \text{vector}(\text{'France'}) + \text{vector}(\text{'Italy'}) \approx \text{vector}(\text{'Rome'})$ ， $\text{vector}(\text{'king'}) - \text{vector}(\text{'man'}) + \text{vector}(\text{'woman'}) \approx \text{vector}(\text{'queen'})$ 。但我们认为这个多少有点被过度炒作了，很多其他降维或主题模型在一定程度上也能达到类似的效果，而且 word2vec 也只是少量的例子完美符合这种加减法操作。

word2vec 大受欢迎的另一个原因是其高效性，Mikolov 在论文 [2] 中指出一个优化的单机版本一天可训练上千亿个词。

21.1 背景

21.1.1 词向量

在 NLP 相关任务中最常见的第一步是创建一个词表库并把每个词顺序编号。这实际上就是词表示方法中的 **One-hot Representation**，这种方法把每个词顺序编号，每个词就是一个很长的向量，向量的维度等于词表大小，只有对应位置上的数字为 1，其他都为 0。

对于这种表示方法，一个最大的问题是无法捕捉词与词之间的相似度，就算是近义词也无法从词向量中看出任何关系。此外，这种表示方法还容易发生维数灾难，尤其是在与深度学习相关的一些应用中。

分布表示最早是由 Hinton 在 1986 年提出的^[3]，其基本思想是：通过训练将每个词映射成 k 维实数向量（ k 一般为模型中的超参数），通过词与词之间的距离（比如 cosine 相似度、欧氏距离等）来判断它们的语义相似度。而 word2vec 使用的就是这种分布表示的词向量表示方式。

21.1.2 统计语言模型

传统的统计语言模型是表示语言基本单位（一般为句子）的概率分布函数，这个概率分布函数也就是该语言的生成模型。一般语言模型可以使用各个词语条件概率的形式表示：

$$p(s) = p(w_1^T) = p(w_1, w_2, \dots, w_T) = \prod_{t=1}^T p(w_t | \text{Context})$$

其中 Context 即为上下文，根据对 Context 不同的划分方法，可以分为以下几大类。

1. 上下文无关语言模型 (Context=NULL)

该模型仅仅考虑当前词本身的概率，不考虑该词所对应的上下文环境。这是一种最简单的但实际应用价值不大的统计语言模型。

$$p(w_t | \text{Context}) = p(w_t) = \frac{N_{w_t}}{N}$$

这种模型不考虑任何上下文信息，仅仅依赖于训练文本中的词频统计。它是 n -gram 语言模型中当 $n = 1$ 时的特殊情形，所以有时也称作 Unigram Model（一元文法统计模型）。在实际应用中，该模型常被应用到一些商用语音识别系统中。

2. n -gram 语言模型 (Context = $w_{t-n+1}, w_{t-n+2}, \dots, w_{t-1}$)

当 $n = 1$ 时，就是上面所讲的上下文无关语言模型，这里 n -gram 一般认为是 $n \geq 2$ 时的上下文相关模型。当 $n = 2$ 时，也称为 Bigram 语言模型，直观地想，在自然语言中“白色汽车”的概率比“白色飞翔”的概率要大很多，也就是 $p(\text{汽车}|\text{白色}) > p(\text{飞翔}|\text{白色})$ 。 $n > 2$ 也类似，只是往前看 $n - 1$ 个词而不是一个词。

一般 n -gram 语言模型优化的目标是最大 Log 似然，即：

$$\sum_{t=1}^T \log p_m(w_t | w_{t-n+1}, w_{t-n+2}, \dots, w_{t-1})$$

n -gram 语言模型的优点是包含了前 $n - 1$ 个词所能提供的全部信息，这些信息对当前词的出现具有很强的约束力。同时，因为只看 $n - 1$ 个词而不是所有词，也使得模型的效率较高。

n -gram 语言模型也存在一些问题。

(1) 高阶 n -gram 语言模型的建立需要大量的语料数据，这在现实中往往无法满足。大部分研究或工作使用的都是 Trigram 或 Bigram。

(2) 这种模型无法建模出词与词之间的相似度，有时候两个具有某种相似性的词，如果一个词经常出现在某段词之后，往往另一个词出现在这段词后面的概率也比较大。比如“白色的汽车”经常出现，那完全可以认为“白色的轿车”也可能经常出现。

(3) 在训练语料里面有些 n 元组没有出现，其对应的条件概率就是 0，导致计算一整句话的概率为 0。

解决这个问题有两种常用方法。

方法一：平滑法。最简单的方法是把每个 n 元组的出现次数加 1，那么原来出现 k 次的某个 n 元组就会记为 $k + 1$ 次，原来出现 0 次的 n 元组就会记为出现 1 次。这种方法也称为 Laplace 平滑。当然，还有很多更复杂的其他平滑方法，其本质都是将模型变为贝叶斯模型，通过引入先验分布打破似然一统天下的局面。而引入先验方法的不同也就产生了很多不同的平滑方法。

方法二：回退法。有点像决策树中的后剪枝方法，即如果 n 元的概率不到，那么就往上回退一步，用 $n-1$ 元的概率乘上一个权重来模拟。

3. n -pos 语言模型 ($\text{Context} = c(w_{t-n+1}), c(w_{t-n+2}), \dots, c(w_{t-1})$)

严格来说， n -pos 只是 n -gram 的一种衍生模型。 n -gram 语言模型假定第 t 个词出现的概率条件依赖于其前 $n-1$ 个词，而现实中很多词出现的概率条件是依赖于其前面词的语法功能的。 n -pos 就是基于这种假设的语言模型，它将词按照其语法功能进行分类，由这些词类决定下一个词出现的概率。这样的词类称为词性 (Part-of-Speech, POS)。 n -pos 语言模型中的每个词的条件概率表示为：

$$p(s) = p(w_1^T) = p(w_1, w_2, \dots, w_T) = \prod_{t=1}^T p(w_t | c(w_{t-n+1}), c(w_{t-n+2}), \dots, c(w_{t-1}))$$

其中， c 为类别映射函数，即把 T 个词映射到 k 个类别 ($1 \leq k \leq T$)。实际上， n -pos 使用了一种聚类思想，不同的词可能属于同一类。原来 n -gram 中每一项 w_i 可能是 T 个单词中的一个，那么 $n-1$ 个单词 $w_{t-n+1}, w_{t-n+2}, \dots, w_{t-1}$ 共有 T^{n-1} 种可能组合，使用类别映射函数 c 后， n -pos 中的每一项 $c(w_i)$ 对应 k 个类别中的一个，那么 $n-1$ 项 $c(w_{t-n+1}), c(w_{t-n+2}), \dots, c(w_{t-1})$ 共有 k^{n-1} 种可能组合。也就是说， n -pos 相对 n -gram 减少了可能的组合数量，同时这种减少还采用了语义有意义的类别。

当然， n -pos 模型还有很多变种，具体可以参考论文 [4]。

4. 基于决策树的语言模型

上面提到的上下文无关语言模型、 n -gram 语言模型、 n -pos 语言模型等，都可以用统计决策树的形式表示出来。而在统计决策树中每个节点的决策规则是一个上下文相关的问题。这些问题可以是“前一个词是 w 吗？”“前一个词属于类别 c_i 吗？”等。当然，基于决策树的语言模型还可以更灵活一些，可以是一些“前一个词是动词吗？”“后面有介词吗？”之类的复杂语法语义问题。

基于决策树的语言模型的优点是：分布数不是预先固定好的，而是根据训练语料库中的实际情况确定的，更为灵活；缺点是：构造统计决策树的问题很困难，且时空开销很大。

5. 最大熵模型

最大熵原理是 E.T. Jayness 于 20 世纪 50 年代提出的, 其基本思想是: 对一个随机事件的概率分布进行预测时, 在满足全部已知的条件下对未知的情况不做任何主观假设。从信息论的角度来说, 就是: 在只掌握关于未知分布的部分知识时, 应当选取符合这些知识但又能使熵最大的概率分布。

$$p(w|\text{Context}) = \frac{e^{\sum_i \lambda_i f_i(\text{Context}, w)}}{Z(\text{Context})}$$

其中 λ_i 是参数, $Z(\text{Context})$ 为归一化因子。因为采用的是 Softmax 形式, 所以最大熵模型有时候也称为指数模型。

6. 自适应语言模型

前面模型中的概率分布都是预先从训练语料库中估算好的, 属于静态语言模型。而自适应语言模型类似于在线学习的过程, 即根据少量新数据动态调整模型, 属于动态模型。在自然语言中, 经常出现这样的现象: 某些在文本中通常很少出现的词, 在某一局部文本中突然大量出现。能够根据词在局部文本中出现的情况动态调整概率分布的语言模型称为动态、自适应或者基于缓存的语言模型。通常的做法是将静态模型与动态模型通过参数融合到一起, 这种混合模型可以有效地避免数据稀疏的问题。

还有一种与主题相关的自适应语言模型, 直观的例子为: 专门针对体育相关内容训练一个语言模型, 同时保留所有语料训练的整体语言模型, 当新来的数据属于体育类别时, 其应该使用的模型就是体育相关主题模型和整体语言模型相融合的混合模型。

21.1.3 神经网络语言模型

神经网络语言模型的英文全称是 Neural Network Language Model, 缩写为 NNLM。在神经网络语言模型方面最值得阅读的文章是深度学习巨头之一——Bengio 发表在 *JMLR 2003* 上的文章^[5]。

NNLM 采用的是分布表示 (Distributed Representation), 即每个词被表示为一个浮点向量。神经网络语言模型如图 21-1 所示。

目标是要学到一个好的模型:

$$f(w_t, w_{t-1}, \dots, w_{t-n+2}, w_{t-n+1}) = p(w_t | w_1^{t-1})$$

需要满足的约束为：

$$f(w_t, w_{t-1}, \cdots, w_{t-n+2}, w_{t-n+1}) > 0$$
$$\sum_{i=1}^{|V|} f(i, w_{t-1}, \cdots, w_{t-n+2}, w_{t-n+1}) = 1$$

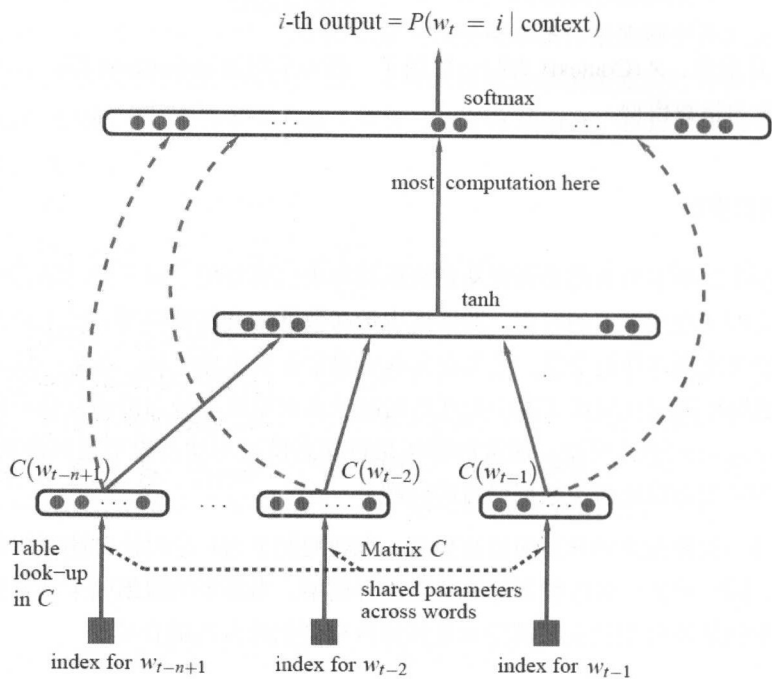


图 21-1 神经网络语言模型

在图 21-1 中，每个输入词都被映射为一个向量，该映射用 C 表示，所以 $C(w_{t-1})$ 即为 w_{t-1} 的词向量。 g 为一个前馈或递归神经网络，其输出是一个向量，向量中的第 i 个元素表示概率 $p(w_t = i | w_1^{t-1})$ 。训练的目标依然是最大似然加正则项，即：

$$\text{Max Likelihood} = \max \frac{1}{T} \sum_i \log f(w_t, w_{t-1}, \cdots, w_{t-n+2}, w_{t-n+1}; \theta) + R(\theta)$$

其中 θ 为参数， $R(\theta)$ 为正则项，输出层采用 Softmax 函数：

$$p(w_t | w_{t-1}, \cdots, w_{t-n+2}, w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}$$

其中 y_i 是每个输出词 i 的未归一化 log 概率, 计算如下:

$$y = b + Wx + U \tanh(d + Hx)$$

其中 b, W, U, d 和 H 都是参数, x 为输入。需要注意的是, 一般神经网络的输入是不需要优化的, 而在这里, $x = (C(w_{t-1}), C(w_{t-2}), \dots, C(w_{t-n+1}))$, 也是需要优化的参数。在图 21-1 中, 如果下层原始输入 x 不直接连到输出的话, 可令 $b = 0, W = 0$ 。

如果采用随机梯度算法, 则梯度更新的法则为:

$$\theta \leftarrow \theta + \eta \frac{\partial \log p(w_t | w_{t-1}, \dots, w_{t-n+2}, w_{t-n+1})}{\partial \theta}$$

其中 η 为学习率。需要注意的是, 一般神经网络的输入层只是一个输入值, 而在这里, 输入层 x 是参数 (存在 C 中), 也是需要优化的。优化结束之后, 词向量有了, 语言模型也有了。

这个 Softmax 模型使得概率取值为 $(0, 1)$, 因此不会出现概率为 0 的情况, 也就是自带平滑, 无需传统 n -gram 模型中那些复杂的平滑算法。Bengio 在 APNews 数据集上做的对比实验也表明, 其模型效果比精心设计平滑算法的普通 n -gram 模型要好 10%~20%。

当然, 神经网络语言模型发展很快, 出现了很多新的变种, 感兴趣的读者可以参考最新的相关论文。

21.1.4 Log-linear 模型

Log-linear 也是 word2vec 所用模型的前身。Log-linear 模型由以下成分组成:

- 一个输入集合 X 。
- 一个标注集合 Y , Y 是有限的。
- 一个正整数 K , 指定模型中向量的维数。
- 一个映射函数 $f: X \times Y \rightarrow \mathbf{R}^K$, 即将任意的 (x, y) 对映射到一个 K 维实数向量。
- 一个 K 维的实数参数向量 $V \in \mathbf{R}^K$ 。

对任意的 x, y , 模型定义的条件概率为:

$$p(y|X; V) = \frac{e^{V \cdot f(x, y)}}{\sum_{y' \in Y} e^{V \cdot f(x, y')}}$$

为何叫 Log-linear? 原因是分子部分取完 log 后是 \mathbf{V} 中各个元素 (特征) 的线性表达式, 例如 $v_1 + v_3 + v_6$ 。Michael Collins^[6] 给出的一个 $K = 8$ 的例子如图 21-2 所示。

$$\begin{aligned}
 f_1(x, y) &= \begin{cases} 1 & \text{if } y = \text{model} \\ 0 & \text{otherwise} \end{cases} \\
 f_2(x, y) &= \begin{cases} 1 & \text{if } y = \text{model and } w_{i-1} = \text{statistical} \\ 0 & \text{otherwise} \end{cases} \\
 f_3(x, y) &= \begin{cases} 1 & \text{if } y = \text{model, } w_{i-2} = \text{any, } w_{i-1} = \text{statistical} \\ 0 & \text{otherwise} \end{cases} \\
 f_4(x, y) &= \begin{cases} 1 & \text{if } y = \text{model, } w_{i-2} = \text{any} \\ 0 & \text{otherwise} \end{cases} \\
 f_5(x, y) &= \begin{cases} 1 & \text{if } y = \text{model, } w_{i-1} \text{ is an adjective} \\ 0 & \text{otherwise} \end{cases} \\
 f_6(x, y) &= \begin{cases} 1 & \text{if } y = \text{model, } w_{i-1} \text{ ends in "ical"} \\ 0 & \text{otherwise} \end{cases} \\
 f_7(x, y) &= \begin{cases} 1 & \text{if } y = \text{model, "model" is not in } w_1, \dots, w_{i-1} \\ 0 & \text{otherwise} \end{cases} \\
 f_8(x, y) &= \begin{cases} 1 & \text{if } y = \text{model, "grammatical" is in } w_1, \dots, w_{i-1} \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

图 21-2 Log-linear 模型中的特征示例

其中 x 是一个文字序列: w_1, w_2, \dots, w_{i-1} , y 为一个词。

可以看出, 各种 n 元语言关系及其变种都可以看作映射函数, 当然 Log-linear 模型的分子需要遍历所有的词汇, 看起来计算量巨大。但因为绝大部分词汇都属于 otherwise 分支, 即为 0, 所以在计算上也是可以接受的。

21.1.5 Log-bilinear 模型

Log-bilinear^[7] 是 Hinton 提出的一种模型, 和 Log-linear 模型的区别在映射函数部分, 之前 $f(x, y)$ 对于一个具体的输入数据都是直接映射到一个 K 维实数向量的, 而在 Log-bilinear 模型中则是直接采用和 y 对应的向量 \mathbf{v}_y 。因为输入的 \mathbf{v} 和 \mathbf{v}_y 都是变量, 所以这种模型取 log 后就变成双线性的了, 因此称为 Log-bilinear。这样带来的问题是分母上的计算变得密集, 当然这难不倒聪明的大佬们, 改进的方法则是引入分类或聚类的思想, 不直接做多分类, 而是每次做二分类, 这就是层次化 Log-bilinear 模型。

21.1.6 层次化 Log-bilinear 模型

Hinton 进一步结合 Bengio 的层次化概率语言模型^[8] 提出了层次化 Log-bilinear 模型^[7]。这种模型的特征是:

- 叶子节点为词 (word) 的二叉树, 而内部节点虽然也有对应的向量, 但并不对应到具体的词。
- 每个节点上都要进行决策。

在 N 元层次化 Log-bilinear 模型中由上下文预测下一个词为 w 的公式为:

$$p(w_n = w | w_{1:n-1}) = \prod_{i=1}^K p(d_i | q_i, w_{1:n-1})$$

其中 d_i 是 w 编码中第 i 位的值, q_i 是该编码对应路径上的第 i 个节点。

$$p(d_i | q_i, w_{1:n-1}) = \sigma(\mathbf{v}_{\text{context}}^T \cdot \mathbf{q}_i)$$

其中 $\mathbf{v}_{\text{context}}$ 是对应上下文的向量, 比如后面将要介绍的 Skip-gram 采用的是输入词对应的向量, 在 CBOW 中是 $w_{1:n-1}$ 对应 $n-1$ 个向量之和, 当然还能带权, 还能加 bias, 还可以换一个更复杂的映射函数, 一个词还能有多个编码。此外, 如何编码也是可以不断创新的, 自然的方法是根据语义分类、聚类等, 也可以选择 Huffman 编码, 也就是 word2vec 的方法。还有, 在每个节点预测时, 除了用 Likelihood (对应伯努利概率分布), 还可以用方差 (对应高斯分布) 等。

word2vec 提出的新的 Log-bilinear 模型包括 CBOW 和 Skip-gram 两种, 具体的介绍请参考下节。

21.2 CBOW 模型

CBOW 是 Continuous Bag-of-Words Model 的缩写, 是一种与前向 NNLM 类似的模型, 不同点在于 CBOW 去掉了最耗时的非线性隐层且所有词共享隐层。如图 21-3 所示, 可以看出, CBOW 模型预测 $P(w_t | w_{t-k}, w_{t-(k-1)}, \dots, w_{t-1}, w_{t+1}, w_{t+2}, \dots, w_{t+k})$ 。

从输入层到隐层所进行的操作实际上就是上下文向量的加和, 具体的代码如图 21-4 所示。其中 sentence_position 为当前 word 在句子中的下标。以一个具体的句子 A B C D 为例, 第一次进入下面的代码时当前 word 为 A, sentence_position 为 0。b 是一个随机生成的 0 到 window - 1 的词, 整个窗口的大小为 $(2 \times \text{window} + 1 - 2 \times b)$, 相当于左右各看 window - b 个词。可以看出, 随着窗口从左往右滑动, 其大小也是随机的, 在 3 (当 $b = \text{window} - 1$ 时) 到 $(2 \times \text{window} + 1)$ (当 $b = 0$ 时) 之间随机变通, 即随机值 b 的大小决定了当前窗口的大小。代码中的 neu1 即为隐层向量, 也就是上下文 (窗口内除自己之外的词) 对应向量之和。

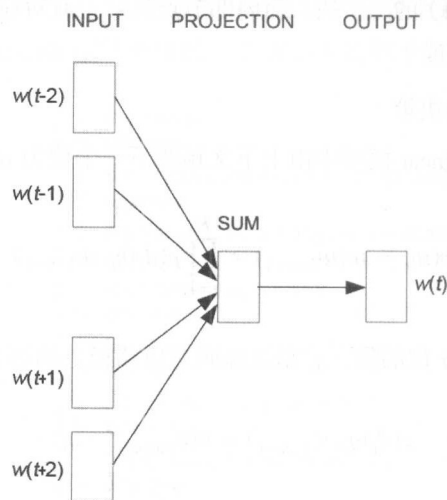


图 21-3 CBOW 模型

```
// in -> hidden
for (a = b; a < window * 2 + 1 - b; a++) if (a != window) {
    c = sentence_position - window + a;
    if (c < 0) continue;
    if (c >= sentence_length) continue;
    last_word = sen[c];
    if (last_word == -1) continue;
    for (c = 0; c < layer1_size; c++) neu1[c] += syn0[c + last_word * layer1_size];
}
```

图 21-4 CBOW 模型对应的部分源码

CBOW 有两种可选的算法：Hierarchical Softmax 和 Negative Sampling。这一部分 Mikolov 没有在论文中进行阐述，下面的相关讨论均来自于本书作者对 word2vec 源码的分析。这种 Hierarchical Softmax 算法结合了 Huffman 编码，每个词 w 都可以从树的根节点沿着唯一一条路径被访问到。这种 Huffman 编码用于神经网络语言模型的方法并不是 word2vec 首创的，作者 Mikolov 在他之前的论文 [9] 和 [10] 中就提到过。假设 $n(w, j)$ 为这条路径上的第 j 个节点，且 $L(w)$ 为这条路径的长度，注意 j 从 1 开始编码，即 $n(w, 1) = \text{root}$ ， $n(w, L(w)) = w$ 。对于第 j 个节点，Hierarchical Softmax 定义的标签为 $1 - \text{code}[j]$ ，这里其实也可以把标签定义为 $\text{code}[j]$ ，得到的向量也差不多，而输出 f 为：

$$f = \sigma(\text{neu1}^T \cdot \text{syn1})$$

loss 为负的 Log 似然，即：

$$\text{Loss} = -\text{Log Likelihood} = -(1 - \text{code}[j]) \log f - \text{code}[j] \log(1 - f)$$

那么梯度为：

$$\begin{aligned} \text{Gradient} &= \frac{\partial \text{Loss}}{\partial \text{neu1}} = -(1 - \text{code}[j]) \cdot (1 - f) \cdot \text{syn1} + \text{code}[j] \cdot f \cdot \text{syn1} \\ &= -(1 - \text{code}[j] - f) \cdot \text{syn1} \end{aligned}$$

$$\begin{aligned} \text{Gradient} &= \frac{\partial \text{Loss}}{\partial \text{syn1}} = -(1 - \text{code}[j]) \cdot (1 - f) \cdot \text{neu1} + \text{code}[j] \cdot f \cdot \text{neu1} \\ &= -(1 - \text{code}[j] - f) \cdot \text{neu1} \end{aligned}$$

CBOW 的第二种算法是 Negative Sampling，这种方法的理论基础是 Noise Contrastive Estimation (NCE)，Negative Sampling 在源代码中随机生成 negative（代码中可设置的变量）个（也有可能少一些，如果随机产生负例时生成了原来的 word）负例，原来的 word 为正例，标签为 1，其他随机生成的标签为 0，输出 f 仍为：

$$f = \sigma(\text{neu1}^T \cdot \text{syn1})$$

loss 为负的 Log 似然（因为采用随机梯度下降法，这里只看一个 word 中的一层），即：

$$\text{Loss} = -\text{Log Likelihood} = -\text{label} \cdot \log f - (1 - \text{label}) \cdot \log(1 - f)$$

那么梯度为：

$$\begin{aligned} \text{Gradient}_{\text{neu1}} &= \frac{\partial \text{Loss}}{\partial \text{neu1}} = -\text{label} \cdot (1 - f) \cdot \text{syn1} + (1 - \text{label}) \cdot f \cdot \text{syn1} \\ &= -(\text{label} - f) \cdot \text{syn1} \end{aligned}$$

$$\begin{aligned} \text{Gradient}_{\text{syn1}} &= \frac{\partial \text{Loss}}{\partial \text{syn1}} = -\text{label} \cdot (1 - f) \cdot \text{neu1} + (1 - \text{label}) \cdot f \cdot \text{neu1} \\ &= -(\text{label} - f) \cdot \text{neu1} \end{aligned}$$

由隐层到输入层的梯度传播则更加简单，因为隐层为输入层各变量的加和，因此输入层的梯度即为隐层梯度。

21.3 Skip-gram 模型

如图 21-5 所示, Skip-gram 模型中的词语指示方向正好与 CBOW 相反, Skip-gram 应该预测概率 $p(w_i|w_t)$, 其中 $t-c \leq i \leq t+c$ 且 $i \neq t$, c 是决定上下文窗口大小的常数, c 越大则需要考虑的对 (pair) 就越多, 一般能够带来更精确的结果, 但是训练时间也会增加。假设存在一个词组序列 w_1, w_2, \dots, w_T , Skip-gram 的目标是最大化:

$$\frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j}|w_t)$$

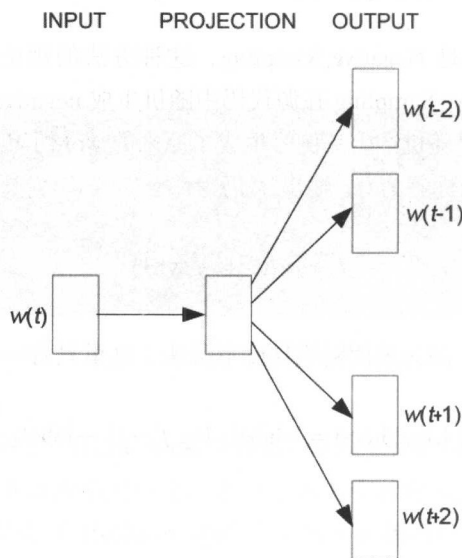


图 21-5 Skip-gram 模型

基本的 Skip-gram 模型定义 $p(w_O|w_I)$ 为:

$$p(w_O|w_I) = \frac{e^{v_{w_O}^T v_{w_I}}}{\sum_{w=1}^W e^{v_w^T v_{w_I}}}$$

从公式不难看出, Skip-gram 是一种对称的模型, 如果以 w_t 为中心词, w_k 在其窗口内,

那么 w_i 也必然在以 w_k 为中心词的同样大小的窗口内，也就是：

$$\frac{1}{T} \sum_{i=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{i+j} | w_i) = \frac{1}{T} \sum_{i=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_i | w_{i+j})$$

同时，Skip-gram 中的每个词向量都表征了上下文的分布。Skip-gram 中的 skip 是指在一定窗口内的词两两都会计算概率，即使它们之间隔着一些词，比如“白色汽车”和“白色的汽车”很容易被识别为相同的短语。

与 CBOW 类似，Skip-gram 也有两种可选的算法：Hierarchical Softmax 和 Negative Sampling。Hierarchical Softmax 算法也结合了 Huffman 编码，每个词 w 都可以从树的根节点沿着唯一一条路径被访问到。假设 $n(w, j)$ 为这条路径上的第 j 个节点，且 $L(w)$ 为这条路径的长度，注意 j 从 1 开始编码，即 $n(w, 1) = \text{root}$ ， $n(w, L(w)) = w$ 。Hierarchical Softmax 定义的概率 $p(w | w_I)$ 为：

$$p(w | w_I) = \prod_{j=1}^{L(w)-1} \sigma(\llbracket n(w, j+1) = \text{ch}(n(w, j)) \rrbracket \cdot v'_{n(w, j)}^T v_I)$$

其中：

$$\llbracket x \rrbracket = \begin{cases} 1, & \text{如果 } x \text{ 为 true} \\ -1, & \text{否则} \end{cases}$$

$\text{ch}(n(w, j))$ 既可以是 $n(w, j)$ 的左子节点，也可以是 $n(w, j)$ 的右子节点，在 word2vec 源代码中采用的是左子节点（标签为 $1 - \text{code}[j]$ ），其实此处改为右子节点也是可以的。

$\text{Loss}_{\text{pair}}$ 为负的 Log 似然（因为采用随机梯度下降法，这里只看一个 pair），即：

$$\begin{aligned} \text{Loss}_{\text{pair}} &= -\text{Log}(\text{Likelihood}_{\text{pair}}) \\ &= -\log(p(w | w_I)) \\ &= -\sum_{j=1}^{L(w)-1} \log(\sigma(\llbracket n(w, j+1) = \text{ch}(n(w, j)) \rrbracket \cdot v'_{n(w, j)}^T v_I)) \end{aligned}$$

对 LR 似然函数比较熟悉的读者应该能看出，该公式与 CBOW 中的对应 loss 公式其实是很类似的，唯一不同是把 $\sigma(\text{neu1} \cdot \text{syn1})$ 变成了 $\sigma(v'_{n(w, j)}^T v_I)$ 。本文根据论文 [2] 的公式推导梯度，因为采用随机梯度下降法，每次只看 w 的一层，假设第 j 层，那么对应的该层

loss 为:

$$\text{Loss} = -\text{Log Likelihood} = -\log \left(\sigma \left(\llbracket n(w, j+1) = \text{ch}(n(w, j)) \rrbracket \cdot v'_{n(w, j)}{}^T v_I \right) \right)$$

(1) 如果 $\llbracket n(w, j+1) = \text{ch}(n(w, j)) \rrbracket$ 为 true, 即当前节点为左子节点, 那么 $\text{loss} = -\log \left(\sigma \left(v'_{n(w, j)}{}^T v_I \right) \right)$ 。

对应的梯度为:

$$\text{Gradient}_{v'_{n(w, j)}} = \frac{\partial \text{Loss}}{\partial v'_{n(w, j)}} = - \left(1 - \sigma \left(v'_{n(w, j)}{}^T v_I \right) \right) \cdot v_I$$

$$\text{Gradient}_{v_I} = \frac{\partial \text{Loss}}{\partial v_I} = - \left(1 - \sigma \left(v'_{n(w, j)}{}^T v_I \right) \right) \cdot v'_{n(w, j)}$$

(2) 如果 $\llbracket n(w, j+1) = \text{ch}(n(w, j)) \rrbracket$ 为 false, 即当前节点为右子节点, 那么 $\text{Loss} = -\log \left(\sigma \left(-v'_{n(w, j)}{}^T v_I \right) \right) = -\log \left(1 - \sigma \left(v'_{n(w, j)}{}^T v_I \right) \right)$ 。

对应的梯度为:

$$\text{Gradient}_{v'_{n(w, j)}} = \frac{\partial \text{Loss}}{\partial v'_{n(w, j)}} = \sigma \left(v'_{n(w, j)}{}^T v_I \right) \cdot v_I$$

$$\text{Gradient}_{v_I} = \frac{\partial \text{Loss}}{\partial v_I} = \sigma \left(v'_{n(w, j)}{}^T v_I \right) \cdot v'_{n(w, j)}$$

(3) 合并 1 和 2 得:

$$\text{Gradient}_{v'_{n(w, j)}} = \frac{\partial \text{Loss}}{\partial v'_{n(w, j)}} = - \left(1 - \text{code}[j] - \sigma \left(v'_{n(w, j)}{}^T v_I \right) \right) \cdot v_I$$

$$\text{Gradient}_{v_I} = \frac{\partial \text{Loss}}{\partial v_I} = - \left(1 - \text{code}[j] - \sigma \left(v'_{n(w, j)}{}^T v_I \right) \right) \cdot v'_{n(w, j)}$$

21.4 Hierarchical Softmax 与 Negative Sampling

本节主要讨论为什么要使用 Hierarchical Softmax 与 Negative Sampling。

前面提到 Skip-gram 中的条件概率为:

$$p(w_O | w_I) = \frac{e^{v_{w_O}^T v_{w_I}}}{\sum_{w=1}^W e^{v_w^T v_{w_I}}}$$

这其实是一个多分类的逻辑回归 (Logistic Regression), 即 Softmax 模型, 对应的标签是 One-hot Representation, 只有当前词对应的位置为 1, 其他为 0。

普通的方法是 $p(w_O|w_I)$ 的分母要对词汇表里的所有单词求和, 这使得计算梯度很耗时。

另一种方法是只更新当前 w_O 、 w_I 两个词的向量, 而不更新其他词对应的向量, 也就是不管归一化项。这种方法也会使得优化收敛很慢。

Hierarchical Softmax 则是介于两者之间的一种方法, 其实是借助了分类的概念。假设把所有的词都作为输出, 那么“橘子”“汽车”都是混在一起的。而 Hierarchical Softmax 则对这些词按照类别进行区分。对于二叉树来说, 则是使用二分类近似原来的多分类。例如给定 w_I , 先让模型判断 w_O 是不是名词, 再判断是不是食物名, 再判断是不是水果, 再判断是不是“橘子”。虽然在 word2vec 论文中, 作者是使用 Huffman 编码构造的一连串两分类。但是在训练过程中, 模型会赋予这些抽象的中间节点一个合适的向量, 这个向量代表了它对应的所有子节点。因为真正的单词公用了这些抽象节点的向量, 所以 Hierarchical Softmax 方法和原始问题并不是等价的, 但是这种近似并不会带来效果上的显著损失, 同时又使得模型的求解规模显著上升。

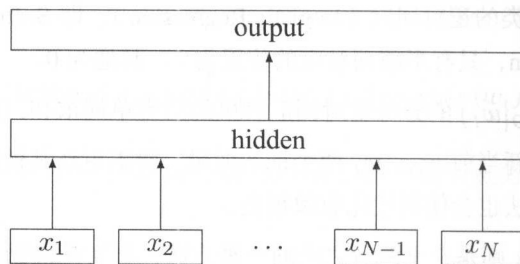
Negative Sampling 也是用二分类近似多分类的, 区别在于它使用的是 one-versus-one 的方式近似, 即采样一些负例, 调整模型参数使得模型可以区分正例和负例。换一个角度来看, Negative Sampling 是一种懒惰的近似方法, 它不想把分母中的所有词都算一次, 就采样一些来计算, 采样多少, 就是模型中负例的个数, 怎么选, 一般按照词频对应的概率分布 (或对高频稍微降权, 比如取平方根) 来随机抽样。

word2vec 的基本原理就介绍到这里, 关于分布式的 word2vec 本文不再赘述。Jeffrey Dean 在论文 [11] 中介绍了异步 SGD 训练大规模深度网络的方法, 完全可以借鉴该论文的思想来实现分布式的 word2vec。

21.5 fastText

word2vec 是一种无监督模型, 而 fastText^[12] 则是对应的有监督模型, 都属于 Tomas Mikolov 的杰作。fastText 由 Facebook 在 2016 年开源, 学习的目标不再是词语内在的共现, 而是人工标注的 label。

fastText 模型架构如图 21-6 所示, 与 word2vec 中的 CBOW 模型类似, 只是中间的单词被替换成了标签。其中 $x_1, x_2, \dots, x_{N-1}, x_N$ 表示一个句子对应的 n -gram 特征, 相比词袋模型, n -gram 特征会关注词语的顺序, 这些特征被转化为词向量并进行平均从而形成隐层变量, 最终的输出层采用 Softmax 计算相应的概率。

图 21-6 fastText 模型架构^[12]

对于 N 个文档的集合，fastText 对应的损失函数为：

$$-\frac{1}{N} \sum_{n=1}^N y_n \log(f(\mathbf{B}\mathbf{A}x_n))$$

其中 x_n 是第 n 个文档对应的归一化统计特征， \mathbf{A} 和 \mathbf{B} 是权重矩阵，这个模型可以在多个 CPU 上使用梯度下降方法并行计算。

与 word2vec 类似，fastText 也采用了层次式的分类器，只是 word2vec 是针对单词的，fastText 则针对 label。

21.6 GloVe

除了 word2vec，LSA（Latent Semantic Analysis）、PLSA（Probabilistic Latent Semantic Analysis）、LDA（Latent Dirichlet Allocation）等传统主题模型也可以用于生成词向量，这种模型对全局的统计信息（比如共现频率、主题等）应用较充分，但是很难体现词与词之间的线性关系（比如：King - Man = Queen - Woman）。而 word2vec 能够较好地体现词与词之间的线性关系，但是对全局的统计信息利用不足。GloVe^[13] 提出的初衷就是为了充分吸收两者的长处。实质上，GloVe 依然是一种矩阵分解方法，非常类似于早期的 LFM（Latent Factor Model）^[14]，只是分解对象变为了共现频率的对数，同时对高低频做了一定的权重调整。

GloVe 的英文全称为 Global Vectors for Word Representation，即单词表示的全局向量，是一种获取单词向量表示的非监督学习方法。GloVe 模型的损失函数为：

$$\text{Loss} = \frac{1}{2} \sum_{i,j=1}^V f(P_{ij})(w_i \tilde{w}_j - \log P_{ij})$$

其中 P_{ij} 表示单词 i 和 j 共现的频率，对于单词共现，其共现频率记为 $1/d$ ， d 为窗口中的距离； $f(P_{ij})$ 表示对权重的调整； $\log P_{ij}$ 也是对高频的降权，以免高频共现词对过度支配模型。实际上模型训练了两套参数： W 及 \tilde{W} ，如果共现矩阵 P 是对称的，则 $W = \tilde{W}$ 。一般来说，共现是对称的，即单词 i 和 j 共现，频率计数对两者来说是一样的，但考虑到模型可能过拟合，一般最终的输出取两者之和或平均。

21.7 小结

总结 word2vec 高效率的原因，本书作者认为主要有以下几点。

- (1) 去掉了费时的非线性隐层。
- (2) Huffman 编码相当于做了一定的聚类，且越高频的计算量越少。
- (3) 采用 Negative Sampling。
- (4) 使用随机梯度算法。
- (5) 只过一遍数据，不需要反复迭代。
- (6) 利用编程实现中的一些技巧，比如指数运算的预计算和高频词亚采样等。
- (7) word2vec 也可以很方便地实现并行化，比如借鉴 Jeffrey Dean 论文 [11] 中的异步 SGD 方法。

参考文献

- [1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.
- [2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.
- [3] Hinton, Geoffrey E. Learning distributed representations of concepts. Proceedings of the eighth annual conference of the cognitive science society. 1986.
- [4] R. Rosenfeld. Two decades of statistical language modeling: where do we go from here?. Proceedings of the IEEE, 88(8), 1270-1288, 2000.
- [5] Bengio Y, Ducharme R, Vincent P, et al. A neural probabilistic language model. Journal of machine learning research, 2003, 3(Feb): 1137-1155.

- [6] Collins M. Log-linear models . Self-published Tutorial. <http://www.cs.columbia.edu/~mccollins/loglinear.pdf>. Accessed December 28, 2015.
- [7] A. Mnih and G. Hinton. Three new graphical models for statistical language modelling. Proceedings of the 24th international conference on Machine learning, pages 641-648, 2007.
- [8] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In Robert G. Cowell and Zoubin Ghahramani, editors, AISTATS'05, pages 246-252, 2005.
- [9] Tomas Mikolov, Stefan Kombrink, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In Acoustics, Speech and Signal Processing (ICASSP), 2011, IEEE International Conference on, pages 5528-5531. IEEE, 2011.
- [10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. ICLR Workshop, 2013.
- [11] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. Proceedings of NIPS, 2012.
- [12] A. Joulin, E. Grave, P. Bojanowski, T. Mikolov. Bag of Tricks for Efficient Text Classification, arXiv preprint arXiv:1607.01759, 2016.
- [13] Pennington J, Socher R, Manning C D. Glove: Global vectors for word representation. EMNLP. 2014, 14: 1532-1543.
- [14] Deepak Agarwal and Bee-Chung Chen. 2009. Regression-based latent factor models. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'09). ACM, New York, NY, USA, 19-28.

神经网络机器翻译

机器翻译作为 NLP 中的一个重要领域，受到了研究人员的广泛关注。从研究方面讲，机器翻译涉及语言的理解和生成，这也是人工智能的目标之一；从工业界讲，机器翻译具有非常可观的商业价值，不同语种之间的人可以通过机器翻译进行交流，各大互联网公司都十分重视并推出了各自的翻译引擎，比如 Google 翻译、百度翻译和 Bing 翻译等。目前机器可以较好地翻译简单的日常用语，对于较为复杂或较为专业的语句，机器翻译也可作为辅助工具，进而提高专业译员的工作效率。

近年来，深度学习，尤其是基于 Attention 的模型，在机器翻译领域取得了长足进展，本章将具体介绍这些相关研究。

22.1 机器翻译简介

早在 1954 年美国 Georgetown 大学和 IBM 公司就合作完成了第一次机器翻译试验，通常将 1954 年到 1990 年这一时期的机器翻译研究方法称为规则机器翻译，顾名思义，就是计算机使用规则进行翻译，每对语种都要花费巨大的人力资源写规则。1993 年，IBM 的研究人员提出了统计机器翻译，不再需要人写翻译规则，而是用计算机通过统计大量的双语平行句对（例如一个中文句子和其对应的英文句子，就是一个平行句对），从而训练出一个翻译模型。一直到 2013 年，在这将近 20 年里，研究人员提出了很多机器翻译的方法，但大多都是以 1993 年 IBM 提出的方法为基础的，我们将这段时期的方法统称为统计机器翻译。统计机器翻译将翻译质量提高到了实用阶段。

2014 年出现了神经网络机器翻译，在经过短短三年的发展之后，效果比发展了 20 年的统计机器翻译好了很多，显示出巨大的潜能。神经网络机器翻译的优势还在于模型简单，实现代码量大概只有统计机器翻译的二十分之一。神经网络机器翻译还实现了端到端（End to End）的模型，极大简化了训练流程。

下面简单介绍统计机器翻译的训练过程，我们以中英文翻译为例。

首先，输入的大量中英文句对，通过一个短语抽取模块，得到一个中英文翻译短语表，形式类似于表 22-1 所示。

表 22-1 中英文翻译短语表

中文短语	英文短语	翻译概率
开发	development	0.5
开发	development of	0.6
举行 会谈	held a talk	0.1
监督 检查	supervisory inspection	0.8
从 上下文 看	from the context	0.3

这里只列举了一个翻译概率，在实际统计翻译中，每对短语有很多概率描述其合法程度。除了翻译短语表，通常还有一个语言模型和一个调序模型。语言模型用来平滑译文，调序模型用来调整短语的顺序。例如，对于“在什么地点干什么事情”的中文句子，其英文译文通常是“do somethine on some place”。对于短语表的概率、语言模型的概率和调序模型的概率，要通过一个训练过程分别给这三个部分一个权重。最终翻译结果就是所有概率加权和价值最大的译文。

比如要翻译一个句子，“布什与沙龙举行了会谈”，统计翻译引擎是这样工作的：

- (1) 分词为“布什 与 沙龙 举行 了 会谈”。
- (2) 查找短语表，找出所有可能的短语对应的英文短语。
- (3) 通过英文短语的翻译概率、语言模型和调序模型，对这些英文短语进行拼接、调序和打分，最终选出得分最高的译文“Bush held a talk with Sharon”。

22.2 神经网络机器翻译基本模型

人类在做中英文句子翻译的时候，首先会理解中文句子表达的意思，然后再以英文的形式表达出来。神经网络机器翻译也是类似的思想，首先用一个编码器（Encoder）将中文句子

编码成一个特征向量,然后再用一个解码器(Decoder)将这个特征向量转换成英文句子。为了叙述简洁,我们以后用 NMT (Neural Machine Translation)^[1] 代替神经网络机器翻译,将待翻译的句子称为源句,对应的译文称为目标句。

编码器就是我们之前介绍过的 RNN 语言模型,解码器也可以看作多了一个输入的 RNN 语言模型。如图 22-1 所示就是 NMT 的架构。

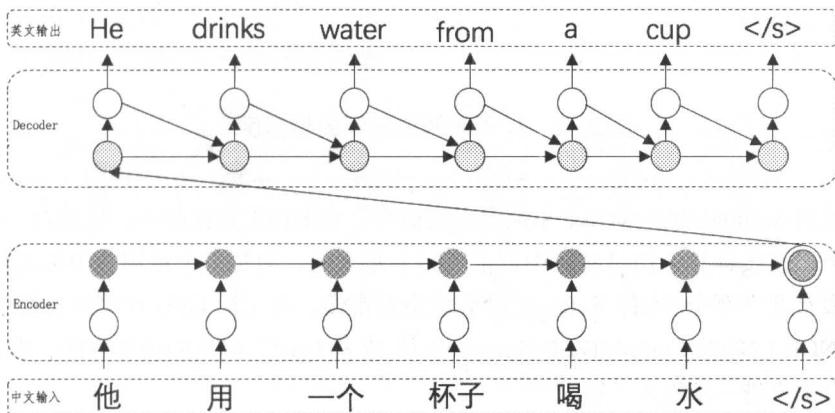


图 22-1 基于 Encoder-Decoder 的 NMT 架构图

将源句输入编码器,对于源句的每个位置的单词,编码器都会生成一个状态值,只取编码器的最后一个位置的状态输出 h_T 作为源句的一个特征向量,也可以认为 h_T 是对源句的一个概括总结。首先,源句中的每个词先用 one-hot 特征向量表示,然后通过 Embedding 层映射为 s_i ,最后状态可以表示为 $h_i = \text{RNN}(h_{i-1}, s_i)$, h_0 通常设为全零向量。

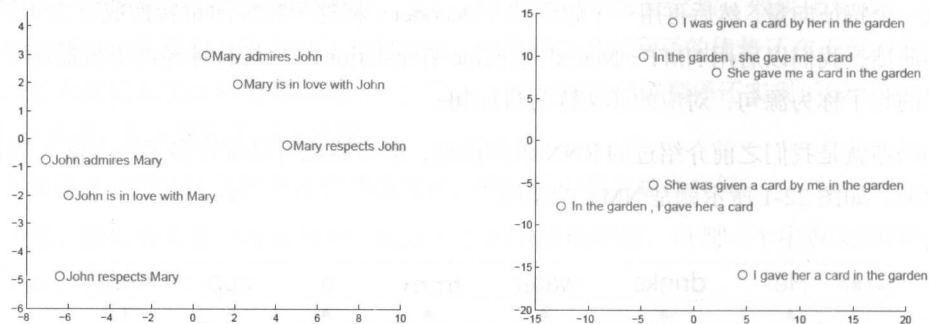
最后的状态 h_T 到底表示什么? Sutskever 等人将高维的 h_T 通过 PCA 映射到两维空间^[2],如图 22-2 所示。从图中可以看到,两个句子越相似,它们的 h_T 距离越近。

解码器和编码器非常类似, z_i 表示解码器的输出状态, u_{i-1} 表示解码器在前一个位置输出的单词(目标句单词),用公式表示为 $z_i = \text{RNN}(h_T, u_{i-1}, z_{i-1})$ 。使用 z_i 计算每个单词出现的概率 p_i :

$$e(k) = w_k^T z_i + b_k$$

$$p(w_i = k | w_1, w_2, \dots, w_{i-1}, h_T) = \frac{\exp(e(k))}{\sum_j \exp(e(j))}$$

在位置 i , 通过概率 p 采样出词 u_i , 再通过公式 $z_{i+1} = \text{RNN}(h_T, u_i, z_i)$ 生成位置 $i+1$ 的词, 重复这样的操作, 直到生成一个结束标记。

图 22-2 句子在两维空间的相似度^[2]

最后使用 Softmax 来产生每个单词出现的概率。如果词汇表比较小，比如在一万个单词以内，就可以直接计算。但是实际中词汇表是非常大的，通常一个可用的中英文翻译引擎，英文词汇表有几十万个单词，Softmax 计算就会非常慢，甚至用 GPU 计算时显存就不够了。通常采用 NCE (Noise Contrastive Estimation)^[3] 或者 Jean^[4] 提出的训练方法，有兴趣的读者可以通过论文进一步深入了解。

22.3 基于 Attention 的神经网络机器翻译

编码器将所有长度的源句都编码成一个固定长度的向量，并不能表示源句的所有细节，特别是在最后位置的词对这个向量的影响会非常大，Bahdanau^[5] 和 Cho^[6] 利用 Attention 改进了 NMT。简而言之，解码器在生成每一个位置的词时，都会从编码器生成的所有的状态值中选择一个状态值作为解码器的输入，而不是像上文所讲的只是选取编码器的最后一个状态值。

对于编码器，希望源句的每个位置的状态值都要包含整个句子的信息，并且这个位置的词对当前的状态值影响最大，所以编码器用双向 RNN 代替了之前的单向 RNN。正向 RNN 在位置 i 的状态值 $\text{forward_}h_i$ 表示从句子开始到位置 i 的词的总结，反向 RNN 在位置 i 的状态值 $\text{backward_}h_i$ 表示从句子结束到位置 i 的词的总结， $h_i = \text{concat}(\text{forward_}h_i, \text{backward_}h_i)$ 则表示整个句子的总结， h_i 受位置 i 的词的影响最大，如图 22-3 所示。

对于解码器，在每个位置都要生成一个权重向量，向量长度就是源句的长度，向量在位置 j 的分量通过下面公式计算：

$$a_j = \frac{\exp(e_j)}{\sum_{j'} \exp(e_{j'})}$$

其中, $e_j = a(z_{i-1}, h_j)$, a 表示 Attention, 具体计算可以参考本书相关章节。直观上理解就是, 通过解码器的上个位置的状态值来选择编码器某个位置的状态值。输入到解码器的特征向量就是编码器的所有状态值的加权和。

$$c_i = \sum_{j=1}^T a_j h_j$$

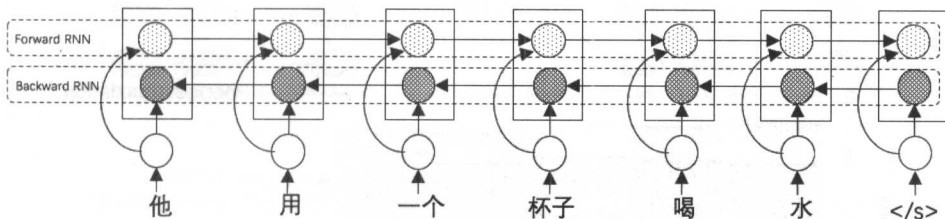


图 22-3 双向 RNN 编码器

引入 Attention 的 NMT 计算架构如图 22-4 所示。

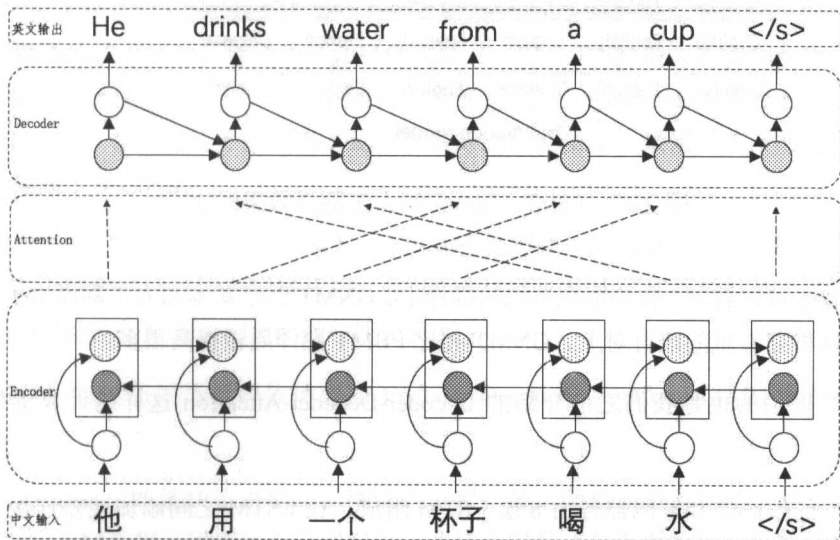


图 22-4 引入双向编码和 Attention 的 NMT 计算架构图

22.4 谷歌机器翻译系统 GNMT

本章第一节提到的统计机器翻译最成熟的模型是基于短语的机器翻译模型（Phase Based Machine Translation, PBMT），在 2015 年之前它还是工业界的主流翻译技术，神经网络技术引入后，在短短的一两年内，NMT 已经全面替换 PBMT。本节主要介绍谷歌的机器翻译系统 GNMT。

首先，我们看一下 PBMT、GNMT 和人工翻译的效果对比，如图 22-5 所示。

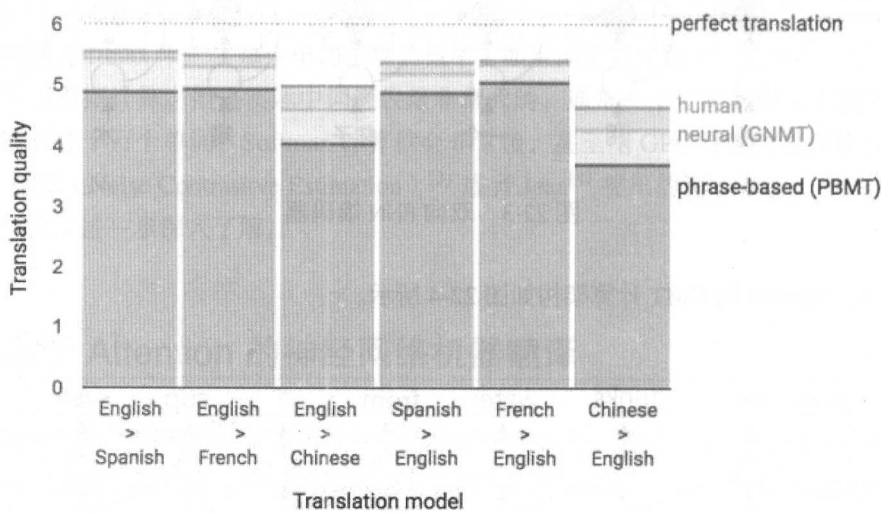
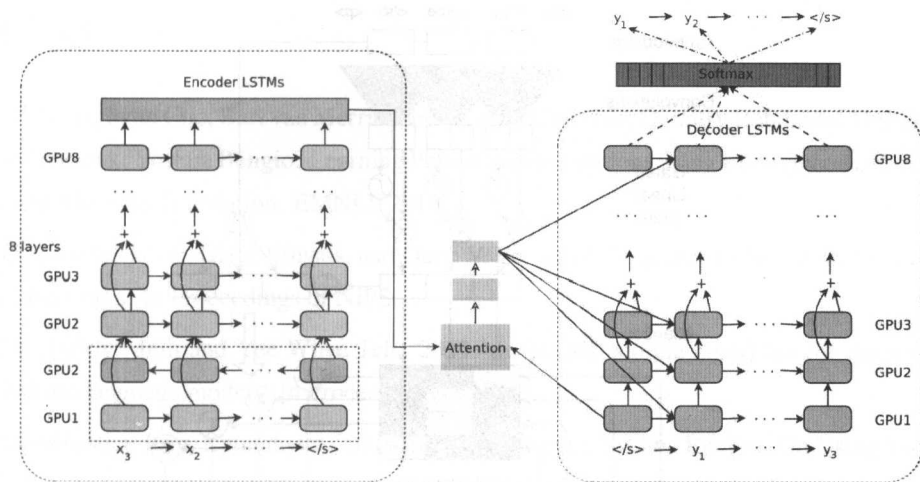


图 22-5 PBMT、GNMT 和人工翻译效果对比^[7]

从图 22-5 可以看到，在法语到英语的翻译上，GNMT 已经基本和人工翻译水平接近，在中英/英中这种非常难的语言对上，GNMT 相比 PBMT 翻译质量提高很多。

GNMT 采用的也是我们之前介绍的 Encoder-Decoder-Attention 这样的计算框架，如图 22-6 所示。

GNMT 的编码器和解码器各由 8 层 LSTM 组成，在 LSTM 之间添加残差连接，这里借鉴的是何恺明提出的应用在图像识别领域的残差网络^[9]。在编码器中，只有第二层 LSTM 是从右向左的，其他 7 层都从左向右的，第二层和第一层组成了双向 LSTM 作为第三层的输入，这样就能更好地利用句子的上下文信息。

图 22-6 GNMT 计算框架图^[8]

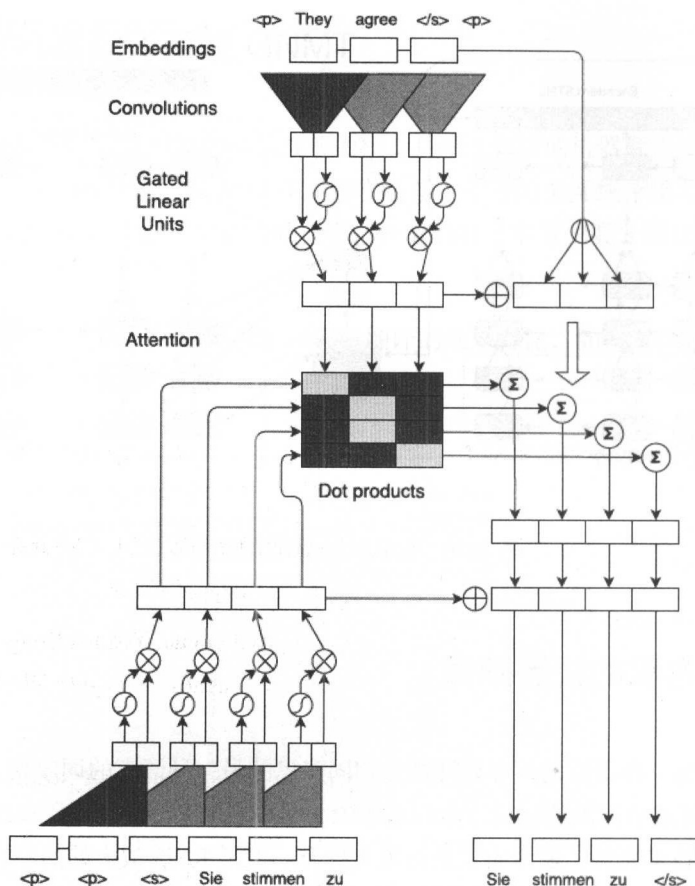
22.5 基于卷积的机器翻译

机器翻译是将一种语言的词序列转换为相同意思的另一种语言的词序列，对于这种序列的操作 RNN 的链式结构天然适用，所以基于 RNN 的神经网络机器翻译在短期内就突破了传统机器翻译的技术瓶颈，翻译质量获得极大提升。但是 RNN 最大的问题在于，对于序列只能逐个处理，无法并行操作，导致很难对速度进行优化。Facebook 的研究者提出用卷积操作代替 RNN^[10]，在翻译任务上取得了比谷歌更好的翻译质量，最值得关注的是在速度上比谷歌翻译提高了 9~20 倍。

卷积在图像识别领域的作用非常大，可以并行处理图片的局部特征，参数少，计算高效。在翻译任务上卷积的这些特性同样适用，通过层叠的卷积构成了层级的结构，这样就能通过较短的卷积层叠路径来获取词之间的长距离关系。

Facebook 的卷积机器翻译依然遵循 Encoder-Decoder-Attention 这样的网络结构，如图 22-7 所示。

图 22-7 中显示的是从英语翻译到德语的过程，最上面的是英语句子，经过 Embedding 层，再经过 15 层卷积，得到用编码器编码成的特征向量；左下面的是解码器部分，将德语句子经过 15 层卷积得到特征向量；中间部分就是根据编码器和解码器得到的特征向量做 Attention。

图 22-7 卷积神经网络机器翻译网络结构图^[10]

22.6 小结

神经网络机器翻译的发展潜力非常大，Google 已经用 NMT 全面替代了之前的统计机器翻译模型。但是 NMT 也存在一些缺点。首先，NMT 的可解释性较低，模型基本是一个黑盒；其次，目前主流的 NMT 系统都会使用多层 LSTM，即使使用 GPU 计算，在超大规模语料上的训练时间也依然比较长，除了做数据并行还需要做模型并行；第三，对于目前已有的语法树等结构性信息和调序信息，NMT 还没有办法很好地融合；第四，NMT 对于不常见词的翻译效果比较弱，比如专有名词、数字等；第五，NMT 有时不会翻译源句的所有部分；最后，NMT 要部署到实际线上环境中提供服务或者部署到移动端，要做很多性能上的优化。

参考文献

- [1] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. EMNLP 2014.
- [2] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In Proceedings of NIPS.
- [3] Andriy Mnih and Yee Whye Teh. 2012. A fast and simple algorithm for training neural probabilistic language models. In Proceedings of ICML.
- [4] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, Yoshua Bengio. On Using Very Large Target Vocabulary for Neural Machine Translation. ACL'15.
- [5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In Proceedings of ICLR.
- [6] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. In Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, 2014.
- [7] A Neural Network for Machine Translation, at Production Scale. <https://research.googleblog.com/2016/09/a-neural-network-for-machine.html>.
- [8] Wu, Yonghui, Schuster, Mike, Chen, Zhifeng, Le, Quoc V, Norouzi, Mohammad, Macherey, Wolfgang, Krikun, Maxim, Cao, Yuan, Gao, Qin, Macherey, Klaus, et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv preprint arXiv:1609.08144, 2016.
- [9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. CVPR 2016.
- [10] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, Yann N. Dauphin. Convolutional Sequence to Sequence Learning. <https://arxiv.org/abs/1705.03122>.

第 5 部分

深度学习研究篇

Batch Normalization

目前，深度神经网络已经广泛应用到各种复杂的机器学习问题中，特别是对于图像、音频、视频等数据，人工提取特征困难，深度神经网络的使用卓有成效。在近几年的研究中，深度神经网络模型的结构逐渐趋向于横向上变窄、纵向上变深的“瘦长网络”。例如，2014 年的 VGGNet^[1]，使用 3 个 3×3 的卷积层代替 1 个 7×7 的卷积层，前者比后者提取的特征更多，而且参数计算量更少；2015 年的 ResNet^[2]，理论上可以达到 1000 层，充分利用模型的深度来提高模型的表征能力。

但是随着神经网络越来越深，训练却越来越难。深度神经网络面临着先天的训练问题，即训练参数难，训练速度慢。实际上，梯度消失问题导致网络层间的梯度无法有效地传递。梯度无法正常地收敛是困扰深度神经网络模型训练的重要难题。

2014 年，Google 的 DeepMind 团队提出了能够对深度神经网络各层次间的输入输出进行批量标准化的算法，有效地解决了多层神经网络梯度消失的问题。通过使用 Batch Normalization，在网络训练过程中各层的梯度变化会趋于稳定，并且能够经过较少的迭代得到较高的精确度。本章主要根据 Batch Normalization（以下简称 BN）论文的主要内容，介绍 BN 的前向和后向传播算法，并讨论 BN 是如何做到加速深度神经网络的训练的，以及 BN 网络模型的优化与实际中的应用方法等问题。

23.1 前向与后向传播

23.1.1 前向传播

Batch Normalization 在前向传播时有三个主要任务。

- 计算出每批训练数据的统计量。
- 对数据进行标准化。
- 对标准化后的数据进行扭转，将其映射到表征能力更大的空间上。

算法 23-1 BN 在 Mini-Batch 上的变换算法

输入：每个 Mini-Batch 上的 x 值 $\mathcal{B} = \{x_1 \dots x_m\}$

需要学习的参数 γ, β

输出： $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

计算 Mini-Batch 均值 $\mu \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$

计算 Mini-Batch 方差 $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$

标准化 $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$

缩放与位移 $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$

1. 从 Mini-Batch 计算均值与方差

SGD 即随机梯度下降算法，与批量梯度下降算法相对应，后者每次都要用到全体的训练数据来迭代更新，前者会根据每个样本数据更新参数，所以 SGD 对参数更新的速度更快，但也更加震荡。为了优化参数更新的方向，SGD 还有一些变式方法比如动量（Momentum）和 Adagrad。

SGD 通过最小化以下损失函数来求解神经网络的最优值：

$$\Theta = \arg \min_{\Theta} \frac{1}{N} \sum_{i=1}^N l(x_i, \Theta)$$

其中 Θ 是神经网络中的参数； $x_1 \dots x_N$ 是训练数据集。

为了综合随机梯度下降和批量梯度下降这两种算法的优点, Mini-Batch 梯度下降在单个样本迭代和全部样本迭代之间找到了一个折中点, 既加快了参数的迭代速度, 也避免了单个样本数据带来的波动性。

当 Mini-Batch 中的数据量为 m 时, 可以通过如下公式计算出梯度:

$$\Delta\Theta = \eta \cdot \frac{1}{m} \sum_{i=1}^m \frac{\partial l(x_i, \Theta)}{\partial \Theta}$$

其中 η 为学习率。

可以证明, 通过小批量样本计算出的梯度可以正确地表示全部训练数据的梯度, 并且每批的数据量越大, 样本梯度越接近于总体梯度。另外, 得益于现代平行计算技术, 使用 Mini-Batch 比 m 次单样本的计算效率更高, 因此比原始的随机梯度下降方法更快。

2. 从批量样本推断总体均值与方差

网络模型在训练阶段与测试阶段使用的统计量是不同的。训练时, 每个 batch 使用本批的统计量来进行标准化, 测试时则需要总体的统计量对数据进行转换。总体统计量可以通过如下公式从每批的样本统计量的移动平均数中推断出来。值得注意的是, 推断方差时需要加上 $m/(m-1)$ 的校正, 因为样本方差均值的 $\frac{m}{m-1}$ 倍才是总体方差的无偏估计。

对于多个 Mini-Batch 的训练集 \mathcal{B} , 每个 batch 大小为 m :

总体均值 $E[x] \leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}]$

总体方差 $\text{Var}[x] \leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$

无偏估计是指估计量的期望值与估计参数的真实值偏差为零, 我们可以通过如下推导证明 $E_{\mathcal{B}}[\mu_{\mathcal{B}}]$ 是 $E[x]$ 的无偏估计, $\frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$ 是 $\text{Var}[x]$ 的无偏估计。

$$E_{\mathcal{B}}[\mu_{\mathcal{B}}] = E\left[\frac{1}{m} \sum_{i=1}^m X_i\right] = \frac{1}{mn} \sum_{i=1}^{mn} X_i = E[x] \quad (\text{此处假设数据有 } n \text{ 个 batch})$$

在推导方差期望之前, 我们首先要了解样本均值的方差公式。

$$\text{Var}[\mu_{\mathcal{B}}] = \text{Var}\left[\frac{1}{m} \sum_{i=1}^m X_i\right] = \frac{1}{m^2} \sum_{i=1}^m \text{Var}[X_i] = \frac{1}{m} \text{Var}[x]$$

$$\begin{aligned}
E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2] &= \frac{1}{m} E \left[\sum_{i=1}^m (X_i - \mu_{\mathcal{B}})^2 \right] = \frac{1}{m} E \left[\sum_{i=1}^m (X_i - E[x] + E[x] - \mu_{\mathcal{B}})^2 \right] \\
&= \frac{1}{m} E \left[\sum_{i=1}^m \left((X_i - E[x])^2 - 2(X_i - E[x])(\mu_{\mathcal{B}} - E[x]) + (\mu_{\mathcal{B}} - E[x])^2 \right) \right] \\
&= \frac{1}{m} E \left[\sum_{i=1}^m (X_i - E[x])^2 - 2 \sum_{i=1}^m (X_i - E[x])(\mu_{\mathcal{B}} - E[x]) + m(\mu_{\mathcal{B}} - E[x])^2 \right] \\
&= \frac{1}{m} E \left[\sum_{i=1}^m (X_i - E[x])^2 - 2m(\mu_{\mathcal{B}} - E[x])^2 + m(\mu_{\mathcal{B}} - E[x])^2 \right] \\
&= \frac{1}{m} E \left[\sum_{i=1}^m (X_i - E[x])^2 - m(\mu_{\mathcal{B}} - E[x])^2 \right] = \text{Var}[x] - \text{Var}[\mu_{\mathcal{B}}] \\
&= \text{Var}[x] - \frac{1}{m} \text{Var}[x] = \frac{m-1}{m} \text{Var}[x]
\end{aligned}$$

3. 数据标准化

数据标准化又叫作数据归一化，是数据挖掘过程中常用的数据预处理方法。当我们使用真实世界中的数据进行分析时，会遇到两个问题：

- 特征变量之间的量纲单位不同。
- 特征变量之间的变化尺度（scale）不同。

特征变量的尺度不同导致参数尺度规模也不同，带来的最大问题就是在优化阶段，梯度变化会产生震荡，减慢收敛的速度。经过标准化的数据，各个特征变量对梯度的影响变得统一，梯度的变化会更加稳定（如图 23-1 所示）。

总结起来，数据标准化有以下三个优点：

- 数据标准化能够使数值变化更稳定，从而使梯度的数量级不会变化过大。
- 在某些算法中，标准化后的数据允许使用更大的步长，以提高收敛的速度。
- 数据标准化可以提高被特征尺度影响较大的算法的精度，比如 k-means、kNN、带有正则的线性回归等。

数据标准化有多种方法，例如 min-max 标准化、z-score 标准化等。

数据标准化的主要计算公式是 $\frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}}$ 。

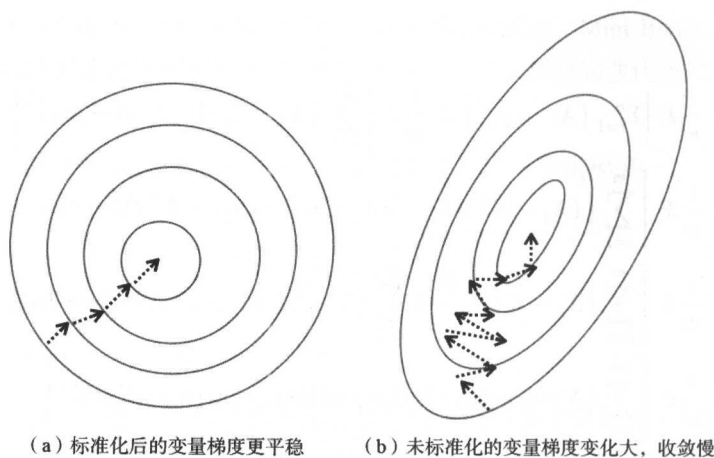


图 23-1 标准化对梯度变化的影响

4. 缩放与偏移 (Scale-Shift)

标准化后的数据还需要进行一定的缩放与偏移等变换，即在标准化后的数据上放大或缩小一定的比例并向上或向下平移一定的距离，变换公式为 $y_i = \gamma \hat{x}_i + \beta$ 。因为此时的数据应该呈均值为 0，方差为 1 的分布，如果激活函数是 Sigmoid 函数，将集中在线性表达的区域，如图 23-2 所示，非线性的表达能力会受到影响。

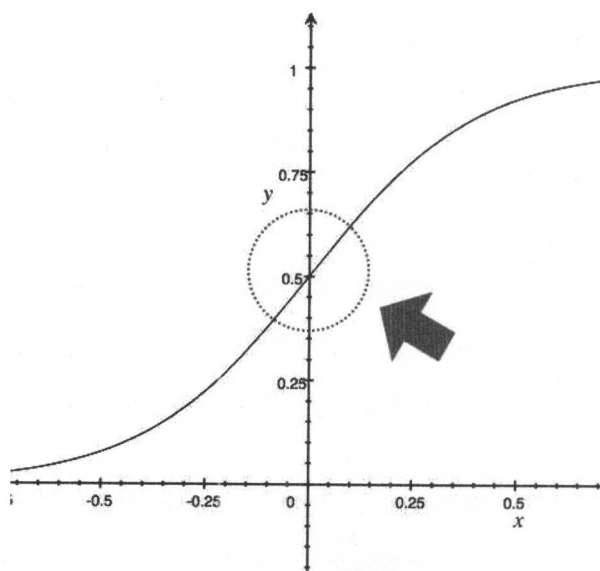


图 23-2 标准化后的数据会降低非线性表达能力

另外,对标准化后的数据进行扭正,也在一定程度上扩大了模型的表征能力,当 $\gamma = \sqrt{\text{Var}[x] + \epsilon}$ 以及 $\beta = \frac{\gamma E[x]}{\sqrt{\text{Var}[x] + \epsilon}}$ 时, $y_i = x_i$, 即数据可以恢复到标准化之前的状态。

23.1.2 后向传播

1. 后向传播过程

后向传播是一种在训练时与梯度下降方法结合使用的优化算法,是训练人工神经网络的常用方法。该方法计算网络中所有权重的损失函数的梯度。计算出的梯度会被送给优化方法,优化方法又使用梯度来更新权重,以试图使损失函数最小化。

通过本节前面的介绍可知,前向传播可以分为三步:将数据标准化、对数据进行线性偏移、将数据输出到下一层,即 $\hat{x}(\mu, \sigma^2, x) \rightarrow y(\hat{x}, \gamma, \beta) \rightarrow l$ (其中, \hat{x} 是标准化后的输入, y 是 \hat{x} 的线性变换, l 代表 BN 的下一层)。同样地,反向传播就是按照相反三步传递误差,即 $l \rightarrow y(\hat{x}, \gamma, \beta) \rightarrow \hat{x}(\mu, \sigma^2, x)$ 。所以求导的过程也依次为 $\frac{\partial l}{\partial y} \rightarrow \frac{\partial y}{\partial \hat{x}}, \frac{\partial y}{\partial \gamma}, \frac{\partial y}{\partial \beta} \rightarrow \frac{\partial \hat{x}}{\partial \sigma^2}, \frac{\partial \hat{x}}{\partial \mu}, \frac{\partial l}{\partial x_i}$ 。具体的计算过程如图 23-3 所示。

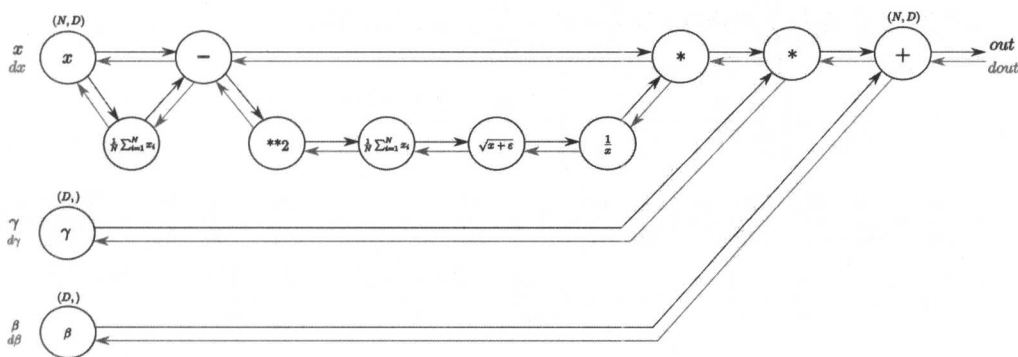


图 23-3 BN 的反向传播过程^[3]

2. 梯度的推导

以下为链式法则的基本公式。

假设 $u = u(x, y)$, $x = x(r, t)$, $y = y(r, t)$, 则 $\frac{\partial u}{\partial r} = \frac{\partial u}{\partial x} \cdot \frac{\partial x}{\partial r} + \frac{\partial u}{\partial y} \cdot \frac{\partial y}{\partial r}$ 。

假设从下一层产生的误差为 l , 则本层产生的误差为 $\frac{\partial l}{\partial y}$ 。

(1) 计算 $\frac{\partial l}{\partial \hat{x}}, \frac{\partial l}{\partial \gamma}, \frac{\partial l}{\partial \beta}$ 。

$$\bullet \frac{\partial l}{\partial \gamma} = \sum_{i=1}^m \frac{\partial l}{\partial y_i} \cdot \frac{\partial y_i}{\partial \gamma} = \sum_{i=1}^m \frac{\partial l}{\partial y_i} \cdot \hat{x}_i$$

$$\bullet \frac{\partial l}{\partial \beta} = \sum_{i=1}^m \frac{\partial l}{\partial y_i} \cdot \frac{\partial y_i}{\partial \beta} = \sum_{i=1}^m \frac{\partial l}{\partial y_i}$$

$$\bullet \frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i} \cdot \frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i} \cdot \gamma$$

(2) 计算 $\frac{\partial l}{\partial \sigma^2}, \frac{\partial l}{\partial \mu}, \frac{\partial l}{\partial x_i}$ 。

$$\bullet \text{ 因为 } \frac{\partial l}{\partial \sigma^2} = \frac{\partial l}{\partial \hat{x}} \cdot \frac{\partial \hat{x}}{\partial \sigma^2}, \text{ 其中, } \hat{x}_i = (x_i - \mu)(\sigma^2 + \epsilon)^{-0.5}$$

$$\text{所以, } \frac{\partial \hat{x}}{\partial \sigma^2} = \sum_{i=1}^m (x_i - \mu) \cdot (-0.5) \cdot (\sigma^2 + \epsilon)^{-0.5-1} = -0.5 \sum_{i=1}^m (x_i - \mu) \cdot (\sigma^2 + \epsilon)^{-1.5}$$

$$\frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^m \frac{\partial l}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\bullet \text{ 因为 } \frac{\partial l}{\partial \mu} = \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial \mu} + \frac{\partial l}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial \mu}, \text{ 其中, } \hat{x}_i = \frac{(x_i - \mu)}{\sqrt{\sigma^2 + \epsilon}}, \frac{\partial \hat{x}_i}{\partial \mu} = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot (-1)$$

$$\text{而且, } \sigma^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu)^2, \frac{\partial \sigma^2}{\partial \mu} = \frac{1}{m} \sum_{i=1}^m 2 \cdot (x_i - \mu) \cdot (-1)$$

$$\text{所以, } \frac{\partial l}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^m \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\bullet \frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial l}{\partial \mu} \cdot \frac{\partial \mu}{\partial x_i} + \frac{\partial l}{\partial \sigma^2} \cdot \frac{\partial \sigma^2}{\partial x_i}, \text{ 其中, } \frac{\partial \hat{x}_i}{\partial x_i} = \frac{1}{\sqrt{\sigma^2 + \epsilon}}, \frac{\partial \mu}{\partial x_i} = \frac{1}{m}, \frac{\partial \sigma^2}{\partial x_i} = \frac{2(x_i - \mu)}{m} x_i$$

$$\text{可以得到, } \frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial l}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

23.2 有效性分析

从前面的算法介绍中可以看出, BN 的原理十分简单, 就是对每批的训练数据进行标准化, 再做适当的线性变换, 但是却可以显著提高深度神经网络的训练效果。一方面, BN 可以有效地减轻神经网络的内部协移 (Internal Covariate Shift); 另一方面, BN 可以有效地改善训练过程中的梯度流的变化, 解决梯度消失的问题, 加快收敛。同时, BN 也在一定程度上起到了 L2 正则的作用。

23.2.1 内部协移

内部协移 (Internal Covariate Shift) 是由于神经网络中每层的输入发生了变化, 造成每层的参数要不断地适应新分布的问题。传统的 Covariate Shift 问题是指经过学习系统后的输入数据的分布发生改变, 是典型的迁移学习的问题。Internal Covariate Shift 与其类似, 数据分布变化的来源从学习系统变成神经网络层。假设有一个两层的神经网络模型, 第一层的映射函数是 F_1 , 第二层的映射函数是 F_2 , 由此得到的输出为: $l = F_2(F_1(u, \Theta_1), \Theta_2)$ 。

于是, 我们可以得到梯度下降的公式:

$$\Theta_2 \leftarrow \Theta_2 - \frac{\alpha}{m} \sum_{i=1}^m \frac{\partial F_2(x_i, \Theta_2)}{\partial \Theta_2}$$

其中, $x = F_1(u, \Theta_1)$, m 依然是批量数据的大小, α 是学习率。

可以看出, 经过两层神经网络之后, 数据已经发生很复杂的变化, 这将导致每层神经元的参数都要不断地调整适应这种输入数据分布的变化, 不仅使网络的收敛速度变慢, 也使得每层超参数的设定变得更加复杂。

BN 可以在数据经过多层神经网络后, 重新回到均值为 0、方差为 1 的分布上, 解决了以上问题, 使数据的变化分布变得稳定, 训练过程也随之变得平稳, 超参数的调整变得简单。

23.2.2 梯度流

梯度流 (Gradient Flow), 是指梯度按照最陡峭的路径逐渐减小的流动变化, 在梯度下降方法中用来描述梯度的变化过程。

1. 梯度去哪儿了

深度神经网络主要是通过反向传播算法来寻找最优解的, 即将梯度从损失函数层向各层反向传递。在传递过程中, 如果梯度没有稳定地下降, 就有可能导致产生梯度爆炸或梯度消失的现象。这个问题是由于梯度通过多层神经网络传递导致最后一层产生级数积累的结果^[4]。假设网络模型中的每一层接收的梯度都是上一层的 K 倍, 那么 L 层神经网络将会产生相对原有输入 K^L 倍的变化, 当 $K > 1$ 时, 最后一层的输入会非常大; 而当 $K < 1$ 时, 最后一层的输入会变得非常小, 比如 $0.9^{10} \approx 0.34867844$ 。所以经过多层的影响后, 前面的神经网络层接收到的梯度可能会趋近于零, 消失了。

传统的网络模型有三种方法来解决梯度消失的问题。

(1) 使用 ReLU 激活函数。Sigmoid 函数在数值过大或过小时都会进入梯度饱和区域, 而且梯度的计算 $\text{output} * (1 - \text{output})$ 衰减得特别快。相对地, ReLU 采用分段激活的方法, 令参数结果变得稀疏, 在激活阶段, 梯度稳定为 1, 有助于网络模型的收敛。ReLU 存在两个问题:

- 网络模型的结果并不是越稀疏越好, 过于稀疏的参数会导致欠拟合。
- ReLU 可能会过早地关闭一些输入的神经元, 使其得不到更新。

(2) 仔细地初始化与调试参数。参数的初始选择对模型训练的影响很大, 适合的初始值可以使模型较快地收敛到理想的结果。但由于深度学习理论尚处于快速发展的阶段, 很多深度神经网络模型的实际应用没有有效的理论解释, 参数的初始化与调试在一定程度上依赖于经验, 仍属于复杂的黑盒问题。

(3) 使用较小的学习率。当使用 Sigmoid 激活时, 较大的学习率会使权重参数变大, 导致层间输入变大, 较大的数值位于激活函数的饱和区域 (如图 23-2 所示), 从而使梯度趋近于零。另外, 学习率属于网络模型中的超参数, 完全依靠外部设置, 也会带来其他的影响。过大的学习率会使梯度产生过多的抖动, 无法收敛; 而较小的学习率会减少参数更新的幅度, 拖慢收敛速度。当使用 ReLU 激活时, 过大的学习率还会使模型在训练初期产生梯度爆炸, 也完全无法收敛。

2. Batch Normalization 带来更好的梯度流

与之前提及的数据标准化的道理类似, BN 能够减少训练时每层梯度的变化幅度, 使得梯度稳定在理想的变化范围内, 从而改善梯度流, 产生以下收益:

- BN 能够减少梯度对参数的尺度或初始值的依赖, 使得调参更加容易。
- BN 允许网络接受更大的学习率, 这是因为 $\text{BN}(Wu) = \text{BN}((aW)u) \Rightarrow \frac{\partial \text{BN}((aW)u)}{\partial u} = \frac{\partial Wu}{\partial u} \Rightarrow \frac{\partial \text{BN}((aW)u)}{\partial aW} = \frac{1}{a} \cdot \frac{\partial Wu}{\partial W}$, 所以学习率的尺度不会明显影响所产生的梯度的尺度。
- 由于梯度流的改善, 模型能够更快地达到较高的精度。

23.3 使用与优化方法

BN 并不是一种新的神经网络模型, 它是与神经网络模型结合使用的数据处理方法。在实际应用时, 可以以独立模块的方式灵活嵌入到神经网络的各层之间, Caffe 的官方版本采

用的就是这种方式。在与卷积层结合时，BN 层一般置于卷积层与激活函数之间。图 23-4 演示的是 ResNet 网络的 Caffe 模型的前几层。

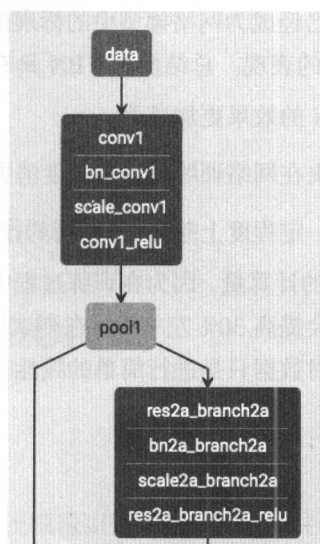


图 23-4 ResNet Caffe 模型的前几层

为了最大化发挥 BN 的优势，在使用 BN 的网络中，可以采用以下几种优化方法。

(1) 增大学习率。在 BN 模型中，增大学习率可以加快收敛速度，但不会对梯度流产生副作用。

(2) 去掉 Dropout。如前所述，BN 可以实现 Dropout 的作用，所以可以去掉 Dropout，以加快训练速度。

(3) 减少 L2 正则的权重。如前所述，BN 有一定的正则作用，所以可以适当地减少 L2 惩罚，参考文献 [5] 在实验中将 L2 正则减少到 1/5，提高了模型在验证集上的表现。

(4) 提高学习率的衰减速度。使用了 BN 后的模型会更快地收敛，所以学习率也应该相应地减小到较低的值。

(5) 更加彻底地随机化训练数据，以防止每批数据总是出现相同的样本。

(6) 减少图片扭曲。因为 BN 的训练速度更快，能够观察到的图片变少了，所以应该让模型尽可能地观察真实的图片。

23.4 小结

在目前发表的论文中，BN 已经成为网络模型中的标准配置，因为它能有效地加快深度神经网络的收敛速度，提高模型的表现。总结前文，BN 具有以下特点。

- 随着网络层数的加深，BN 的效果更加显著。
- BN 可以改善梯度流，解决在网络训练过程中梯度消失的问题。
- BN 可以减轻过拟合，在一定程度上起到 L2 正则的作用。

但是 BN 可能会加重训练时的计算量，因为在训练过程中 BN 要不断地更新均值与方差，如果逐层使用 BN，那么计算量会提高 30% 左右^[4]。在测试推断时，由于均值和方差采用训练数据的全局统计量，所以 BN 对数据只是进行简单的线性变换，并不会影响预测时的速度。

参考文献

[1] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.

[2] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[J]. arXiv preprint arXiv:1512.03385, 2015.

[3] <https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>.

[4] Mishkin D, Matas J. All you need is a good init[J]. arXiv preprint arXiv:1511.06422, 2015.

[5] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[J]. arXiv preprint arXiv:1502.03167, 2015.

在 CNN、FCN、RCNN 等传统图像识别模型中，输入往往是一整张图，然后使用以卷积为主的模型迭代出结果。Google DeepMind 的科学家指出，这种方式具有明显的不足。如果图像中要识别的目标过多、信息过大，RCNN 之类模型的区域检测加识别的方式往往很慢，因为每一个 ROI (Region Of Interest) 都要先判定是否是候选框，如果是候选框，还要送去预测子网络做预测。尽管近年来的论文不断改进卷积模型，让大量卷积层实现共享以提升效率，但传统模型的总体效率和精度也难以再大幅提高了。

近年来对神经学和认知科学的研究表明，传统模型效率不高（准确率往往也不够高）的原因很可能在于其观察事物的方式不够好。既然我们想要解决 ROI 在哪里并且是什么，那么何必要在全图上均匀探索信息（卷积），而不直接从一个 ROI 跳转到另一个呢？Attention 机制据此另辟蹊径，提供了一种全新的认知图像的方式^{[1]~[5]}。

我们先来看看人是怎么辨识一张图上的信息的。人在看东西的时候往往会首先定位一个感兴趣的区域，这种该关注哪里就看哪里的机制就是注意力机制（Attention）。辨识出该区域的信息后，目光再移动到下一个感兴趣的区域，对于细节丰富、信息量大的地方，目光可能移动缓慢；对于背景等不重要的地方，可能只稍微看看甚至直接跳过；对于看不懂的地方，目光可能先跳过，最后再折回来（双向 Attention）。目光按序扫完整张图后，人对图上的信息也就基本了解了。

Attention 仿照上述观察方式，在深度学习网络中加入了感兴趣区域的移动和定位机制，使比较复杂的任务分解成相对简单的子步骤序列。每一步只关注特定小区域，抽取出区域表征的信息，再整合到之前的步骤所积累的信息中。在这样反复观察和整合中，模型逐步学懂

了输入源中所蕴含的信息（甚至是层次关系），并对自己的判断也越来越有信心。同时，图片中信息分类置信度的变化，也作为一个反馈信号进一步刺激模型去学习怎样定位下一步的观察位置。

24.1 从简单 RNN 到 RNN + Attention

近年来 RNN 模型的爆炸式发展使传统模型的整体输入（如整张图）分解为序列化的子输入（整张图的小块）变为可能。简单的 RNN 模型可以在积累多步的时序信息之后得出结论，但是这种迭代方式存在的问题。每一步 RNN 的输出向量 $h(t)$ 一般只依赖于上一步的输出 $h(t-1)$ ，这样随着步数的增加，离 t 时刻较远的信息会被慢慢丢失。LSTM 解决了长距离信息丢失的问题，它拥有一个记忆区，通过三个门开关（输入门、遗忘门、输出门）对记忆区进行维护。每当有新的输入到来时，LSTM 就将新的信息有选择地加入到记忆区里，并对以往的信息进行有选择的保留，将记忆区的新老信息整合之后再选择性输出。但是这种方法也有明显的不足，即 LSTM 每一步接受的输入只能是事先给定的，因此这个输入可能是冗余信息，甚至是对当前 t 时刻不恰当的信息。基于 Attention 的模型则强调从之前输入已观察到的结果（从输入中观察到的信息）和下一步如何部署观察位置之间的互动，其采用的 Attention 机制就是根据已观察到的环境信息从众多备选输入中选择“最合适”的输入。当 t 时刻输入 $z(t)$ 到来时，会得到一个输出向量 $h(t)$ ，同时 RNN 的内部信息状态也将被改变。输出向量 $h(t)$ ，可以代表模型当前大脑中存储的对环境信息的表征（Encoded Vector），此时将反过来作为 Attention 模块的输入，用以计算出下一步 RNN 模型的输入 $z(t+1)$ 。Attention 与 RNN 的关系如图 24-1 所示。

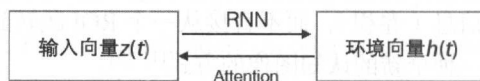


图 24-1 Attention 与 RNN 的关系

24.2 Soft Attention 与 Hard Attention

Attention 模块所进行的选择过程可以表现为某种函数映射 f ，具体实现分为两种形式：Soft Attention 和 Hard Attention。如果所有候选都以一定的概率参与选择，即结果为所有候选向量的概率加权，则称为 Soft Attention；如果只从候选中选取某一个候选向量，则称为 Hard Attention。

举个例子，图片经过多次卷积后得到类似于图 24-2 所示的一个网格，其中 a 、 b 、 c 、 d 对应于经过卷积之后从 4 个位置抽取出的候选向量。那么如何通过映射函数 f 得到下一步输入到 RNN 中的向量 $z(t+1)$ 呢？首先，Attention 模块会在当前步根据 RNN 的环境向量 $h(t)$ 算出一个候选向量的概率分布，如图 24-3 所示。对于 Soft Attention，求 z 需要将每个候选向量进行概率加权，即 $z = p_a a + p_b b + p_c c + p_d d$ ；Hard Attention 则仅仅通过图 24-3 所示的概率分布采样出一个候选向量赋值给 $z(t+1)$ 。总的来说，Soft Attention 和 Hard Attention 机制的主要区别如表 24-1 所示。

a	b
c	d

图 24-2 候选向量

p_a	p_b
p_c	p_d

图 24-3 候选向量对应的概率

表 24-1 Soft Attention 与 Hard Attention 机制的主要区别

策略	Soft Attention	Hard Attention
筛选范围	全部	一个
计算方式	概率加权	采样
是否可导	可导	不可导
训练方式	梯度下降	强化学习

24.3 Attention 的应用

本节以 MNIST 字符串识别为例来讲解 Attention 的具体应用。OCR 的传统做法是首先对字符串进行切分，然后将所得到的单字符送往 CNN 等网络进行识别，最后将识别结果拼接起来。但 Attention 提供了一种全新的思路，如图 24-4 所示的 MNIST 数字串，就可以不通过切分而直接使用 Attention 分步识别。

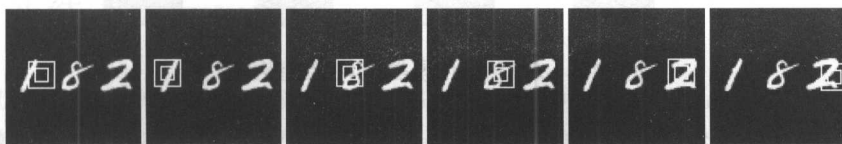


图 24-4 MNIST 的全新识别方式

如何做到呢？模型每一步只把 Attention 集中在当前的小框（称为 Glimpse）所表现的像素和位置信息上，并以每两步一个字符的方式进行检测。小框从左到右扫过，获得的信息通过 RNN 不断累积到环境信息（模型的记忆区）中。可以看到，这样的观察模式在直观上和人观察事物的方式几乎一致。RNN 就像人的视觉中枢，负责迭代和整理信息，Glimpse 就像人的视网膜负责捕捉信息。

具体的网络结构如图 24-5 所示，大致可分为五部分，分别为上下文（Context）模块、一瞥（Glimpse）模块、发射（Emission）模块、循环（Recurrent）模块和分类（Classification）模块。这里采用的是 Hard Attention 模式（一次只采样选择一个小框）^[3]。由于选取小框位置的发射模块引入了随机采样，使得模型无法使用传统的梯度下降法进行训练。然而，一种叫做“策略梯度”（Policy Gradient）的强化学习技术使得训练定位小框位置变得可能。在每一次探索中，模型都会选择某位置序列 $\{(x_1, y_1), (x_2, y_2) \dots\}$ 。如果经过该路径后，分类模块能正确预测出图片中的字符，发射模块就会得到一个正反馈，强化对这个位置序列的选择；反之，如果最后没有预测对，则会得到一个负反馈，告知发射模块需要选择别的路径。在工程实现中，这个反馈的数值通常以一定的比例追加到原损失函数的后面，再整合起来利用传统的梯度下降法训练模型。

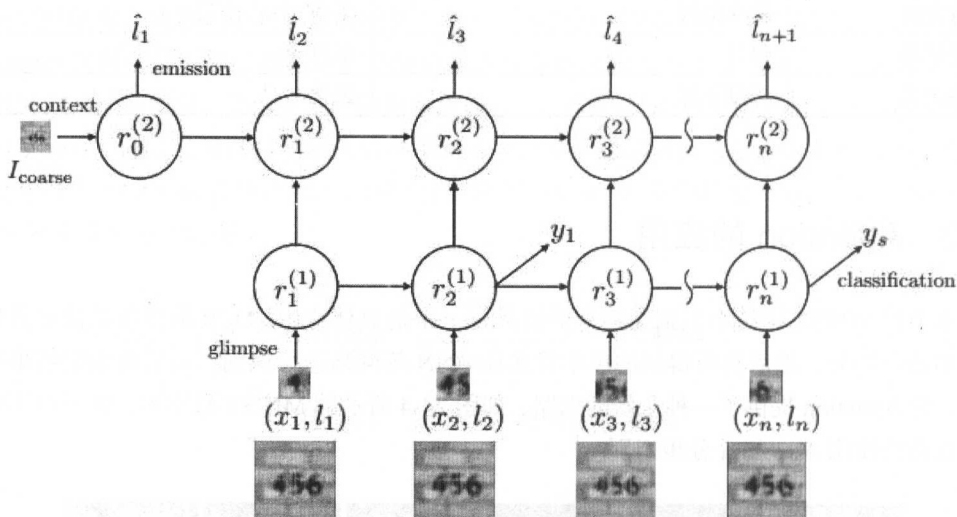


图 24-5 Attention 模型的网络结构图^[1]

下面详细介绍每个模块的功能，以及 Attention 机制是如何在神经网络各个模块之间进行交互的。

1. 一瞥模块

一瞥模块接受表示位置的二维向量 (x, y) 作为输入, 以该位置为中心画出一个框 (Glimpse), 然后对框包含的信息进行提取 (可使用 CNN 提取成特征图, 也可直接使用框的原生像素), 最后再结合框的位置信息, 整合得到一个中间向量 z 。这个向量表征了模型“瞥一眼”图片所能获得的信息总和。

2. 循环模块

循环模块接受从 Glimpse 中提取出的中间向量 z 作为输入。随着多个 Glimpse 的不断积累, 模型就逐步掌握了图片中存储的信息。该迭代模块包含两层 RNN, 下层主要积累识别信息 (认出了什么), 上层则主要负责预测位置 (下一步该往哪里看)。

3. 发射模块

将上层 RNN 输出的累积位置信息 (通常是隐层的高维向量) 映射成二维位置向量, 并以这两个位置为中心点进行高斯采样, 得到下一步 Glimpse 的中心位置 (x, y) 。

4. 分类模块

采用 Softmax 根据下层 RNN 的输出向量进行预测。

5. 上下文模块

上下文模块主要解决第一步该往哪里看的问题。该模块接受整张图作为输入, 输出一个初始状态向量作为上层 RNN 的初始输入, 从而得到第一个 Glimpse 的位置。

24.4 小结

可以看到, 近年来越来越多领域的论文都在应用模型中引入了 Attention。众多实验结果表明, 相比于直接扫描整张图片, Glimpse 序列可以更好地捕捉图片的信息, 从而使目标效果更优。这种将输入分解成序列化的, 同时学习 “Where & What” (在哪里以及是什么) 的思想就是 Attention 模型的核心。由于 Soft Attention 可以直接通过梯度下降法进行训练, 和传统的神经网络训练方法并无太大的区别, 在此不再举例详述。论文 [5] 就使用了 Soft Attention,

其模型模拟人画画的方式，一笔一笔地画出数字，每一次只把 Attention 集中在画一小块的地方，画完一小块再跳转到另一小块，如图 24-6 所示。

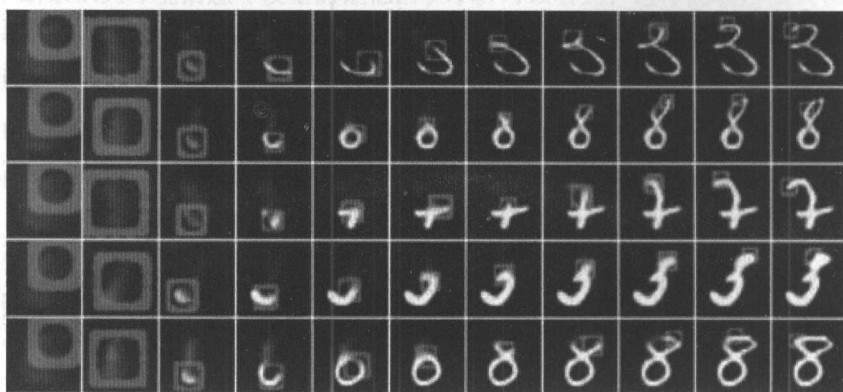


图 24-6 基于 Soft Attention 的数字书写过程^[5]

参考文献

- [1] Jimmy Ba, Volodymyr Mnih, Koray Kavukcuoglu. Multiple Object Recognition with Visual Attention. ICLR 2015.
- [2] Xu K, Ba J, Kiros R, et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. ICML. 2015.
- [3] Volodymyr Mnih, Nicolas Heess, Alex Graves, Koray Kavukcuoglu. Recurrent Models of Visual Attention, NIPS 2014.
- [4] CS231n Course. http://cs231n.stanford.edu/slides/winter1516_lecture13.pdf.
- [5] Gregor, Karol, Danihelka, Ivo, Graves, Alex, and Wierstra, Daan. Draw: A recurrent neural network for image generation. arXiv preprint arXiv:1502.04623, 2015.

25

多任务学习

多任务学习 (Multi-Task Learning, MTL) 是一种采用归纳迁移机制的机器学习方法, 主要目标是利用隐含在多个相关任务的训练信号中的特定领域信息来提高泛化能力。多任务学习通过使用共享表示并行训练多个任务来完成这一目标^[1]。一言以蔽之, 多任务学习在学习一个问题的同时, 可以通过使用共享表示来获得其他相关问题的知识^[2]。归纳迁移学习 (Inductive Transfer Learning) 是一种专注于将解决一个问题的知识应用到相关问题上的方法, 从而提高学习的效率。比如, 学习行走时掌握的能力可以帮助学会跑, 学习识别椅子的知识可以用到识别桌子的学习中, 我们可以在相关的学习任务之间迁移通用的知识。

25.1 背景

在 1997 年之前, 很多归纳学习系统仅专注于单一任务, 因此归纳迁移领域的研究者提出了这个问题: 如何通过使用多个学习任务的知识来增强学习能力^[3]? 归纳迁移学习可以在减少训练样本数目或训练迭代次数的情况下, 达到同等水平的性能, 这对于训练样本不足的学习任务而言是非常有意义的。1997 年, 《机器学习》杂志发表了一个关于归纳迁移的特刊^[4], 归纳迁移领域的主要研究者发表了一系列文章研究如何训练多个学习任务的问题。其中参考文献 [1] 提出了一种学习多个任务的算法并介绍了多任务学习的过程, 在三个领域展现了多任务学习的能力, 以及讨论了该算法可以应用的场景。实验结果表明, 在现实中有很多可以应用多任务学习的场景。

首先,多任务学习可以学到多个任务的共享表示,这个共享表示具有较强的抽象能力,能够适应多个不同但相关的目标,通常可以使主任务获得更好的泛化能力。此外,由于使用了共享表示,多个任务同时进行预测时,减少了数据来源的数量,以及整体模型参数的规模,使预测更加高效。因此,在多个应用领域中,可以利用多任务学习来提高效果或性能,比如垃圾邮件过滤、网页检索、自然语言处理、图像识别、语音识别等。

既然同时学习多个相关任务有重要的意义,那么什么是相关任务?有些理论对任务的相关性刻画已经很清楚了:

- 如果两个任务是处理输入的相同函数,但是在任务信号中加入独立的噪声处理,那么很明显这两个任务是相关的。
- 如果两个任务用于预测同一个个体的属性的不同方面,那么这些任务比预测不同个体的属性的不同方面更相关。
- 两个任务共同训练时能相互帮助并不意味着它们是相关的。例如,通过后向传播网络的一个额外输出中加入噪声可以提高泛化能力,但是这个噪声任务与其他任务不相关。

随着深度学习被广泛应用,计算机视觉和语音识别领域也有了更深远的发展。深度学习网络是具有多个隐层的神经网络,逐层将输入数据转化成非线性的、更抽象的特征表示。并且在深度学习网络中各层的模型参数不是人为设定的,而是给定学习器的参数后在训练过程中学到的,这给了多任务学习施展拳脚的空间,具备足够的能力在训练过程中学习多个任务的共同特征。

25.2 什么是多任务学习

图 25-1 展示了 4 个独立的神经网络,每个网络都是一个针对同样输入仅有一个输出的函数。误差反向传播被应用于这些网络来单独训练每个网络,由于这些网络相互之间没有任何连接,因此其中一个网络学习到的特征并不能帮助另一个网络学习。本文称这种方法为单任务学习 (Single Task Learning, STL)。

图 25-2 展示了一个输入与图 25-1 中的 4 个网络一致的单一网络,但该网络有 4 个输出,每个输出对应于图 25-1 中的一个任务。需要注意的是,这些输出可以连接它们共享的一个隐层的所有神经元,也可以如图 25-2 所示,在共享的一个隐层后形成独立的子网络,训练不与其他任务共享的参数。本文称这种方法为多任务学习。在多任务学习网络中,后向传播并行地作用于 4 个输出。由于 4 个输出共享底部的隐层,这些隐层中用于某个任务的特征表

示也可以被其他任务利用，促使多个任务共同学习。多个任务并行训练并共享不同任务已学到的特征表示，是多任务学习的核心思想。

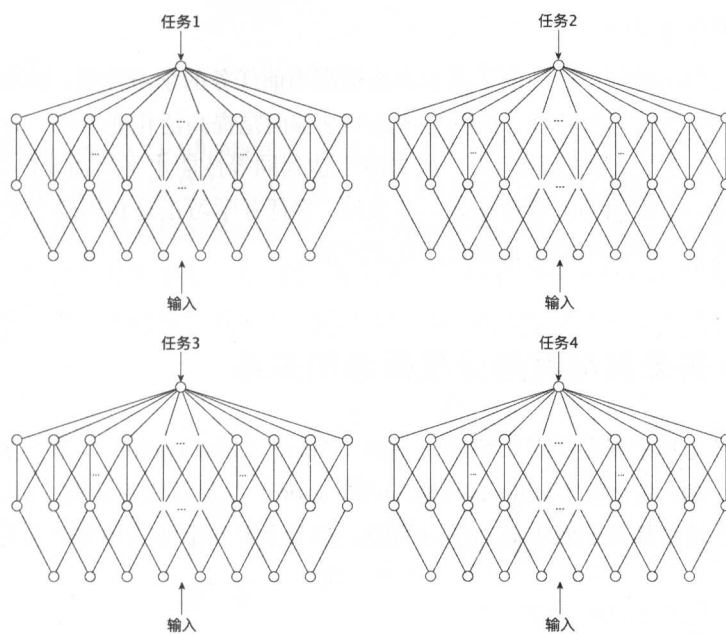


图 25-1 4 个输入相同的任务的单任务后向传播

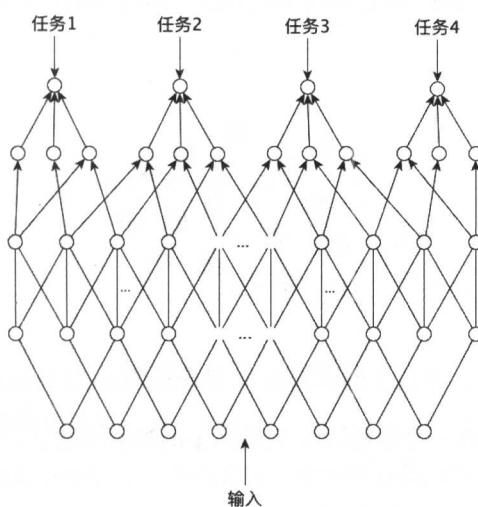


图 25-2 4 个输入相同的任务的多任务后向传播

多任务学习是一种归纳迁移方法，充分利用了隐含在多个相关任务训练信号中的特定领域信息。在后向传播过程中，多任务学习允许共享隐层中专用于某个任务的特征被其他任务使用；这样多任务学习将可以学习到可适用于几个不同任务的特征，这样的特征在单任务学习网络中往往不容易学到。

归纳迁移的目标是利用额外的信息来源来提高当前任务的学习性能，包括提高泛化准确率、学习速度和模型的可解释性。提供更强的归纳偏向是提高泛化能力的一种方法，可以在固定的训练集上产生更好的泛化能力，或者减少达到同等性能水平所需要的训练样本数量。归纳偏向会导致一个归纳学习器更偏好一些假设，多任务学习正是利用隐含在相关任务训练信号中的信息作为一个归纳偏向来提高泛化能力的。

25.3 多任务分类与其他分类概念的关系

在机器学习尤其是深度学习领域，二分类（Binary Classification）、多分类（Multi-Class Classification）、多标签分类（Multi-Label Classification）以及多任务分类（Multi-Task Classification）是关系紧密却又截然不同的学习问题，本节主要介绍前三种分类及其关系。

25.3.1 二分类

二分类（Binary Classification）是最常见的分类问题，比如判断一张图片是不是人脸。如图 25-3 所示为二分类示例，其中不同颜色表示不同的类别，中间的斜线为对应的线性分类面。

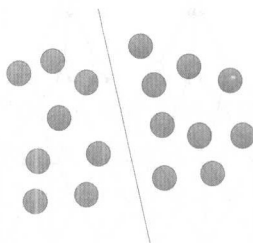


图 25-3 二分类示例

在数学上，即是要选择一个从输入 x 到输出 y 的映射函数 f ：

$$f: x \rightarrow y$$

一般情况下：

$$x \in \mathcal{R}^d, y = \{0, 1\}$$

25.3.2 多分类

多分类 (Multi-Class Classification), 顾名思义, 是指类别数量大于 2 的分类问题。比如常见的中文或英文 OCR 问题是多分类问题; 在目标检测中针对每个候选框的分类是多分类问题。在这类问题中, 总的标签数量大于 2, 且每条数据只对应于其中一个标签。比如识别问题, 输入的图片必须对应于车、人、猫、未知等多个标签中的一个。如图 25-4 所示为三分类示例, 类似地, 其中不同颜色表示不同的类别, 中间的斜线为对应的线性分类面。

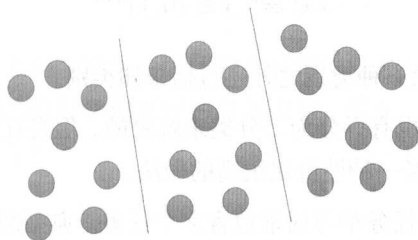


图 25-4 三分类示例

在数学上, 即是要选择一个从输入 x 到输出 y 的映射函数 f :

$$f : x \rightarrow y$$

一般情况下：

$$x \in \mathcal{R}^d, y = \{0, 1, \dots, k\}$$

25.3.3 多标签分类

多标签分类 (Multi-Label Classification) 是指输入源可以含有多个物体, 标注的标签也有多个, 比如图 25-5 对应的标签有: 猫、狗。

在数学上, 则是将输入 x 映射为一个向量 y , 而非多分类问题中的标量 y :

$$f : x \rightarrow y$$

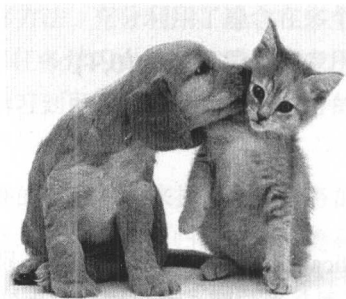


图 25-5 多标签分类的输入图片

一般情况下：

$$x \in \mathcal{R}^d, y \in \{0, 1\}^{k+1}$$

y 为 $k+1$ 维，是因为除 k 种类别之外还包括未知类别。

传统的多标签分类问题也有拆分为二分类来解决的，但是在深度学习盛行的今天，全连接层后套多个 Logistic 输出是一种性价比很高的做法。

根据 25.2 节的定义，多任务学习通常包含多个任务。典型的例子是 Fast RCNN，同时做 Softmax 分类和 bbox 的回归，这是两个完全不同的任务。如果多任务学习中的每个任务都为二分类问题，那么这种多任务分类问题就是一种广义上的多标签分类问题。

一般来说，多任务学习中不同任务之间的区别较大，所以往往不共享所有层，比如 Fast RCNN 就是独立的两个全连接层（Fully Connected Layer, FC），根据任务之间的区别大小，可以决定在不同的层开始分道扬镳。这种底层共享、高层分道扬镳的做法就是迁移学习（Transfer Learning）。

25.3.4 相关关系

二分类、多分类、多标签分类、多任务学习、迁移学习的相互关系如图 25-6 所示。

- 二分类是 $n=2$ 时的多分类。
- 多分类是多标签分类的一种，是对多个相互独立的标签进行学习。
- 多标签学习是多任务学习的一种，每个任务对应一个标签。
- 多任务学习是迁移学习的一种，迁移学习中的源领域和目标领域对应多任务学习中的不同任务。

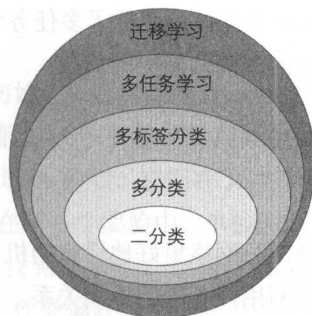


图 25-6 多任务学习与多个概念之间的关系

25.4 多任务学习如何发挥作用

为什么在一个神经网络上同时训练一个任务及其相关任务的学习效果会更好？这是因为额外任务的信息有助于共享的隐层学到更好的内部表示。那么，多任务学习是如何利用相关任务的训练信号中的信息的呢？

25.4.1 提高泛化能力的潜在原因

有很多潜在原因解释了为什么为后向传播网络增加额外的输出可以提高泛化性能。比如，论文 [5] 中通过偶尔为后向传播网络增加噪声来增强泛化能力。

在某种程度上，不相关任务对聚合梯度的贡献对于其他任务来说可以视为噪声。那么，不相关任务也可以通过作为噪声源来提高泛化能力。

另一种可能性是增加任务会影响网络参数的更新。比如增加额外的任务提高了隐层有效的学习率，具体取决于每个任务输出的错误反馈权重。

还有一种可能性是网络的容量，多任务网络在所有任务之间共享网络底部的隐层，或许使用更小的容量就可以获得同水平或更好的泛化能力。

可以设计实验来验证这些可能性的解释，揭示多任务学习可受益于与主任务相关的任务的训练信号。论文 [1] 中做了这样一组实验：准备一个训练集，对于训练集中的每一个样例，包含一组输入特征、主任务训练信号和一组额外任务的训练信号。在所有样例中打乱额外任务的训练信号，但保持额外任务的其他属性以及分布不变。如果多任务学习依赖于训练信号中与主任务相关的额外信息，那么打乱顺序将会削弱这种关系，进而减少多任务学习的收益；如果多任务学习的收益不依赖于多个输出的某些其他属性，那么打乱顺序将不会影响

最终的收益。实验结果是打乱额外任务的顺序降低了多任务学习的性能，这表明观察到的多任务学习的收益应归功于额外任务的训练信号中的信息。

25.4.2 多任务学习机制

这一节总结了几种帮助多任务学习网络更好地泛化的机制，它们都衍生于对不同任务在隐层的误差梯度求和，但是各自又利用任务间的不同关系。

1. 统计数据增强

相关任务的训练信号中的额外信息相当于有效地增加了样本大小，特别是训练信号中存在噪声时，相当于做了数据增强。假定有两个任务 T 和 T' ，在它们的训练信号中都加入了独立的噪声，都受益于计算隐层中输入的特征 F 。一个同时学习了 T 和 T' 的网络，如果发现两个任务共享 F ，则可以使用两个训练信号通过在不同的噪声处理过程中平均化 F ，从而更好地学习 F 。

2. 属性选择

考虑两个任务 T 和 T' ，使用共同的特征 F ，假定网络有很多个输入，如果训练数据比较有限或噪声比较明显，那么一个学习 T 的网络有时难以区分与 F 相关和不相关的数据。然而，一个同时学习 T 和 T' 的网络将可以更好地选择与 F 相关的属性，因为数据增强为 F 提供了更好的训练信号，使该网络能更好地判断哪些输入用于计算 F 。属性选择是数据增强的结果。

3. 信息窃取

考虑一个对任务 T 和 T' 都有用的隐层特征 F ，该特征在学习 T 时比较容易学到，而在学习 T' 时比较难学到（因为 T' 用一种更复杂的方式使用 F ，或者没有 F 时 T' 学到的残留误差更大）。一个网络学习 T 将可以学到 F ，但一个只学习 T' 的网络将不能做到。如果一个网络在学习 T' 时还学习 T ，那么 T' 可以在隐层中窃取 T 已经学到的信息，因此可以学得更好。一旦 T' 和 F 的表示建立了连接，就可以根据任务 T' 相对于 T 独有的额外信息进一步学习 F 。

4. 表示偏置

因为神经网络以随机权重初始化，后向传播是一个随机搜索的过程，多次运行很少产生同样的网络。如图 25-7 所示，假定任务 T 能够发现两个极小值区域 A 和 B ，一个学习任务 T' 的网络同样有两个极小值区域 A 和 C ，它们在 A 共享极小值，但在 B 和 C 没有重叠区域。论文 [1] 中做了两个实验，在第一个实验中，单独训练 T 的网络以均等概率找到 A 和 B ，单独训练 T' 的网络以均等概率找到 A 和 C 。而同时训练 T 和 T' 的网络，通常两个任务都只找到了 A 。这表明多任务学习任务偏好其他任务也偏好的隐层表示，搜索对 T 或 T' 单独学习到的表示的交集倾斜。

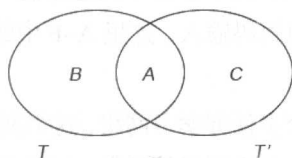


图 25-7 相关任务极小值区域重叠示意图

在第二个实验中，选择使 T 对 B 有强烈偏好的极小值：一个训练 T 的网络总是落入 B 。然而， T' 依然对 A 或 C 没有倾向。当 T 和 T' 同时在一个网络上训练时， T 如预料中落入 B ； T' 未能帮助 T 从 B 拉入 A 。令人惊讶的是， T' 通常落入 C ，未与 T 共享极小值。 T 在隐层表示中创建一个趋势从 A 流向 B ，而 T' 对 A 或 C 没有任何倾向，但受制于 T 创建的趋势，因此通常落入 C 。多任务学习任务倾向于不使用其他任务不偏好使用的隐层表示。

25.4.3 后向传播多任务学习如何发现任务是相关的

多任务学习后向传播网络能否发现任务是否相关？答案是肯定的。后向传播网络虽然主要用于监督学习，但可以在不同任务学习的隐层特征上完成有限的非监督学习，揭示后向传播网络如何发现任务之间的相关性非常有意义。

论文 [1] 中设计了一系列测试问题，称为峰值函数，每个峰值函数的形式如下：

IF (\$1 > ½) THEN \$2, ELSE \$3

其中 \$1、\$2 和 \$3 从 {A、B、C、D、E、F} 中取 3 个字母的排列进行实例化，共有 120 个函数：

```

P001=IF(A > ½) THEN B, ELSE C
P002=IF(A > ½) THEN B, ELSE D
...
P014=IF(A > ½) THEN E, ELSE C
...
P024=IF(B > ½) THEN A, ELSE F
...
P120=IF(F > ½) THEN E, ELSE D

```

A~F 这些变量是定义域为 $[0, 1]$ 的实数，作为一个后向传播网络学习峰值函数的输入。A~F 的值以编码的方式给神经网络，而不是简单地连续输入。一个学习峰值函数的网络不仅需要学习函数，还必须学习正确地解码输入。这里 A~F 中的每个值都使用 10 个输入来编码，总共有 60 个输入。

在所有 120 个函数上训练一个多任务学习网络，这个网络有 60 个输入和 120 个输出，每个输出对应 120 个函数中的一个。分析网络的参数，可以观察到多少个不同的输出共享了隐层，论文 [1] 中对每个输出和每个隐藏单元都做了敏感度分析。通过对比输出 P001 对每个隐藏单元和输出 P002 对每个隐藏单元的敏感度，可以测量 P001 和 P002 共享隐层的程度。结果表明，两个峰值的相关性取决于它们使用几个相同的变量，以及它们是否以同样的方式使用这些变量。比如 P001 和 P120 不共享任何变量，因此它们是不相关的。P001 和 P024 共享了两个变量，但以不同的方式使用它们；P001 和 P014 也共享了两个变量，并且以同样的方式使用它们。因此 P001 与 P014 比与 P024 更相关。通过更加深入的分析，可以发现在条件 IF 测试的重叠比在 THEN 或 ELSE 部分的重叠更重要。

25.5 多任务学习被广泛应用

25.5.1 使用未来预测现在

通常有价值的特征在做出预测之后才变得可用，这些特征在运行时不可用，因此不能作为输入。如果可以离线学习的话，那么这些特征可以收集起来作为训练集，在额外的任务中使用。学习器为这些额外任务做出的预测在系统被使用时可能会被忽略，这些预测的主要功能是为学习器在训练时提供额外的信息。

一个从未来学习的应用是医疗风险预测，比如肺炎风险问题，在这个问题中，可以使用可用的实验室测试作为额外的输出任务，但在为病人诊断时这些测试是不可用的。来自于未来测量方法中的有价值的信息将会帮助网络学习可以更好地支持风险预估的隐层表示。

未来测量方法在很多离线学习问题中可用，有些特征不能在运行时实时获得，因此不能作为输入。但作为多任务学习的输出，它们提供了可以帮助学习的额外信息，即使这些输出在运行时不能获得。

25.5.2 多种表示和度量

有时获得一个误差度量或一种输出表示中比较重要的一切是很困难的。当可替代的度量（或输出表示）可捕获一个问题的其他有用的信息时，多任务学习可以从中获益。

25.5.3 时间序列预测

时间序列预测这种类型的应用是用未来预测现在的一种分类，未来任务和现在任务是相同的，除了它们发生在更迟的时候。

预测的最简单方式是使用一个有多个输出的单一网络，每个输出与在不同时间的相同任务对应。图 25-2 展示的是一个有 4 个输出的神经网络，如果输出 k 指的是任务在时间 T_k 的预测，那么该网络可以对相同任务在 4 个不同的时间做出预测。

25.5.4 使用不可操作特征

有些特征在运行时使用是不现实的，因为它们的计算代价太高，或者因为它们需要不可利用的人类专门知识。训练集通常比较小，并且我们会花费大量的时间来准备，为训练集计算的不可操作特征可以用作多任务的输出。

一个很好的例子是场景分析，通常需要人类知识来标注重要的特征，而学习系统在使用时人不会参与其中。那么这意味着人标注的特征就不能用于学习吗？当然不是，如果训练集可以获得这些标注数据，那么它们能够作为额外任务来使用，但是在系统使用时这些额外任务将不是必要的。

25.5.5 使用额外任务来聚焦

学习器通常倾向于使用输入中大量的、普遍存在的模式，而忽视少量的或不常见的但有用的输入。多任务学习可以通过强制学习来支持依赖严重但可能会被忽略的输入。

25.5.6 有序迁移

有时我们根据之前的学习已经有了相关任务的模型，而用于训练这些模型的数据可能不再可用了。多任务学习可以在没有训练数据的条件下从这些预训练的模型中受益吗？我们可以使用模型来生成人造数据，并在人造数据中使用训练信号作为额外任务。

25.5.7 多个任务自然地出现

通常这个世界给予我们一系列的相关任务可供学习，传统的方法是将它们划分成独立的问题各自训练；但是相关任务一起训练的话能够使彼此受益。一个很好的例子是语音识别需要学习单词的音素和重音。

25.5.8 将输入变成输出

在很多机器学习应用中，有些特征作为输入是不现实的，多任务学习提供了一种通过使用这些特征作为额外任务并从中获益的方式。有些特征作为输出比作为输入会更好，我们可以构造一些问题，使得特征作为输出比输入更有用。

假定有下面的函数：

$$F1(A,B) = \text{sigmoid}(A+B), \text{sigmoid}(x) = \frac{1}{1+e^{(-x)}}$$

考虑如图 25-8 所示的后向传播网络，有 20 个输入、16 个隐藏单元和 1 个输出，通过训练来学习 $F1(A,B)$ 。 $F1(A,B)$ 的数据是通过在区间 $[-5,5]$ 随机采样 A 和 B 的数值生成的，用输入中的 10 位二进制编码表示 A 和 B 。目标输出是 $F1(A,B)$ 的实数值。

表 25-1 展示了三个后向传播网络进行 50 次试验的平均性能，对于每一次试验，都重新生成随机训练集、验证集和测试集。

表 25-1 单任务学习、有额外输入的单任务学习和多任务学习在 $F1$ 函数上的性能对比

网络	试验次数	平均 RMSE
STD（没有额外输入、输出）	50	0.0648
STD+IN（有额外输入）	50	0.0647
MTL（有额外输出）	50	0.0631

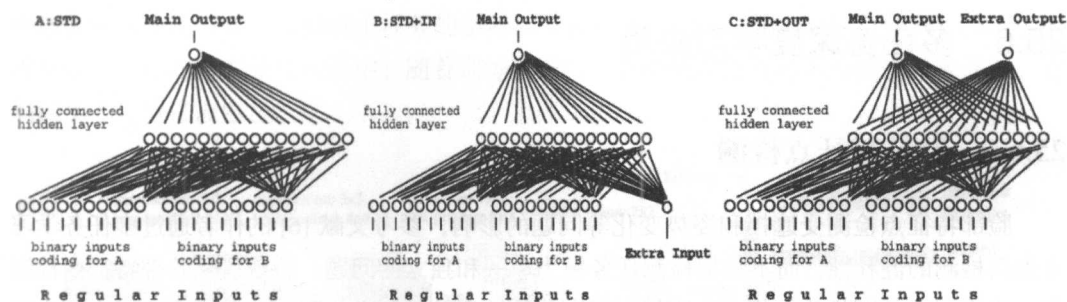


图 25-8 三个学习 $F1$ 的网络架构 (A:STD 是一个不使用额外特征的标准网络; B:STD+IN 是一个使用额外特征作为额外输入的网络; C:STD+OUT 是多任务学习网络, 额外特征作为额外输出使用^[1])

现在考虑 $F1$ 的相关函数:

$$F2(A, B) = \text{sigmoid}(A - B)$$

假设除了用于 A 和 B 的 20 个输入, 还多一个输入给 $F2(A, B)$ 作为额外输入特征, 如图 25-8 (b) 所示。这个额外输入会帮助 $F1(A, B)$ 学得更好吗? 不一定! 对于随机数值 A 和 B , $A + B$ 和 $A - B$ 并不相关, 对于生成的训练集, 这两个随机变量的相关系数的绝对值小于 0.01。这将削弱后向传播网络学习使用 $F2(A, B)$ 来预测 $F1(A, B)$ 的能力。表 25-1 中的 STD+IN 说明了增加额外输入的单任务学习网络的性能, 相比没有额外输入, 性能没有显著变化。可见, 将特征作为输入时, 包含在特征 $F2(A, B)$ 中的额外信息并不会帮助后向传播网络学习 $F1(A, B)$ 。

如果使用 $F2(A, B)$ 作为额外输入不会帮助后向传播网络学习 $F1(A, B)$, 那么是否可以忽略 $F2(A, B)$ 呢? 当然不可以, 因为 $F1(A, B)$ 和 $F2(A, B)$ 是密切相关的, 它们都需要从二进制编码中解码出子特征 A 和 B 。如果将 $F2(A, B)$ 作为必须学习的额外输出, 那么它将使共享的隐层更好地学习 A 和 B , 因此最终将帮助网络更好地学习预测 $F1(A, B)$ 的能力。

图 25-8 (c) 展示了一个有 20 个输入和 2 个输出的网络, 其中一个输出用于 $F1(A, B)$; 另一个输出用于 $F2(A, B)$ 。只在 $F1(A, B)$ 的输出上对网络做性能评估, 但在两个输出上都做后向传播。表 25-1 中的最后一行 MTL 说明了多任务学习网络在 $F1(A, B)$ 上的平均性能, 使用 $F2(A, B)$ 作为额外输出提高了 $F1(A, B)$ 的性能。在这个问题上, 使用额外特征作为额外输出比作为额外输入好。

有一类特别有趣的问题是当特征中存在噪声时, 将特征作为输出比作为输入更有用, 因为额外输出中的噪声比额外输入中的噪声危害小。

25.6 多任务深度学习应用

25.6.1 脸部特征点检测

脸部特征点检测受遮挡和姿势变化等问题的影响，参考文献 [6] 的作者通过多任务学习来提高检测的健壮性，而不是把检测任务视为单一和独立的问题，希望优化脸部特征点检测和一些不同但细微相关的任务，比如头部姿势估计和脸部属性推断。这是有意义的，因为不同的任务有不同的学习困难和收敛率。为了定位这个问题，他们构想出一个新的限制任务的深度模型，通过尽早停止辅助任务来促使学习收敛。

脸部特征点检测不是一个独立的问题，它的预测会被一些不同但细微相关的因素所影响。比如一个正在笑的孩子会张开嘴，有效地发现和利用这个相关的脸部属性将帮助更准确地检测嘴角。如图 25-9 所示，TCDCN 除了检测特征点任务，还有识别眼镜、笑脸、性别和姿态这 4 个辅助任务，通过与其他网络的对比，可以看出辅助任务使主任务的检测更准确。

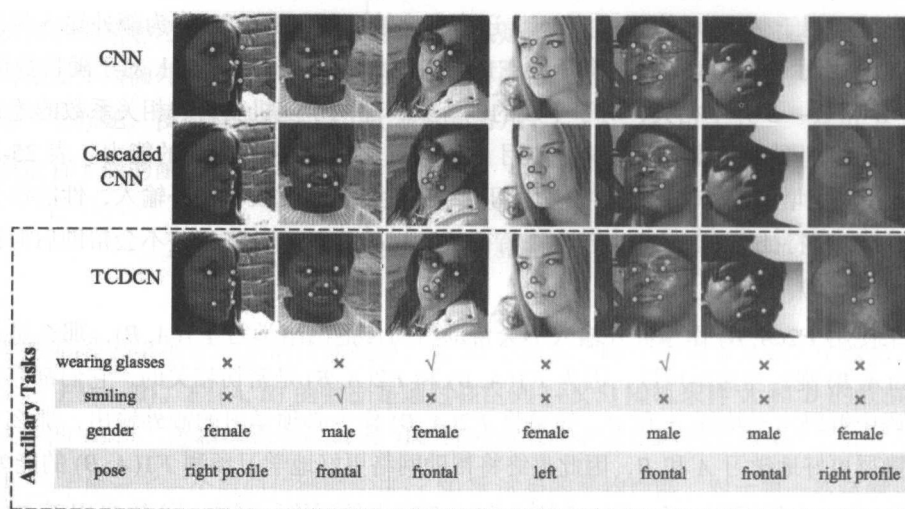


图 25-9 卷积神经网络、级联卷积神经网络和任务约束深度卷积网络（TCDCN）的脸部特征点检测例子^[6]

TCDCN 通过随机梯度下降法进行学习，不同的任务有不同的损失函数和学习难度，因此有不同的收敛速度。其网络结构如图 25-10 所示。早期的方法通过探索任务之间的关系，比如通过学习所有任务权重的协方差矩阵来解决收敛速度不同的问题，但是这种方法只能在所有任务的损失函数相同时才能应用。TCDCN 采用一种尽快停止辅助任务的方法，避免这

些任务对训练集过拟合后影响主任务的学习——在训练开始时，TCDN 受所有任务的约束，避免陷入不好的局部最优状况中；随着训练的进行，有些辅助任务将不再使主任务受益，它们的学习过程将被停止。

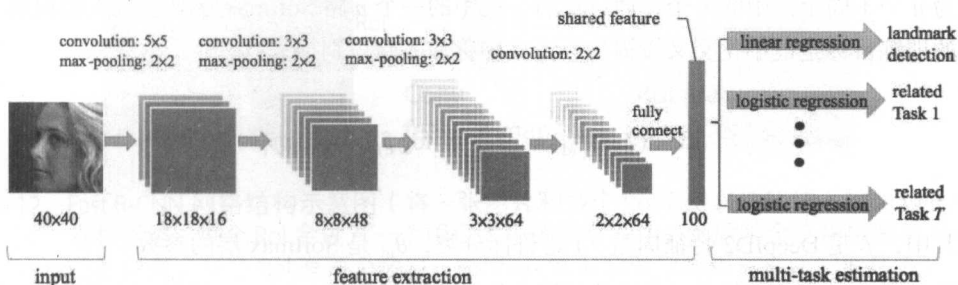


图 25-10 TDCN 网络结构示意图^[6]

25.6.2 DeepID2

论文 [7] 中提出了一种同时使用人脸识别和人脸验证作为监督的方法，以面对在扩大个体之间的差异的同时减少个体内部变化的有效特征的挑战。DeepID2（Deep Identification-verification）通过精心设计的深度卷积网络来学习，人脸识别任务通过刻画从不同个体提取的 DeepID2 特征来增加个体之间的差异，而人脸验证任务通过激励从相同个体提取的 DeepID2 特征来减少个体内部的变化。在 LFW^[8] 数据集上达到 99.15% 的人脸验证准确度。DeepID2 的网络结构如图 25-11 所示。

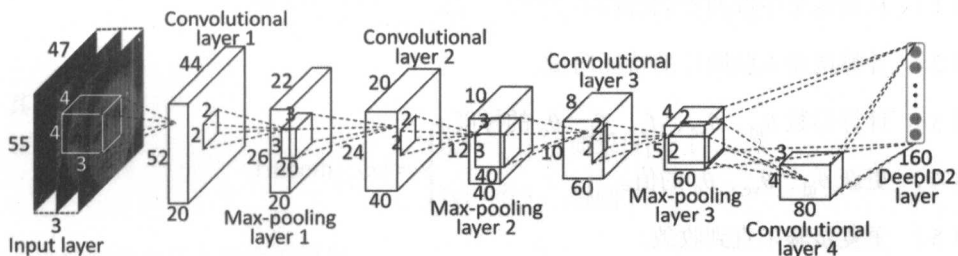


图 25-11 DeepID2 特征提取卷积网络结构示意图^[7]

DeepID2 网络中包含 4 个卷积层，在最后一层（DeepID2 层）提取一个 160 维的 DeepID2 特征向量。DeepID2 特征提取过程可以表示为：

$$f = \text{Conv}(x, \theta_c)$$

其中, x 是输入的人脸图片, f 是提取的 DeepID2 特征向量, θ_c 是卷积网络将要学习的参数。

DeepID2 特征是通过两个监督信号来学习的。第一个是人脸识别信号, 把每张人脸图片归类为 n 个不同个体中的一个, 通过最后一层中的一个 n 路 Softmax 层来实现人脸识别。该网络的训练目标是最小化交叉熵损失, 论文中称作人脸识别损失:

$$\text{Ident}(f, t, \theta_{\text{id}}) = - \sum_{i=1}^n p_i \log \hat{p}_i = - \log \hat{p}_t$$

其中, f 是 DeepID2 特征向量, t 是目标分类, θ_{id} 是 Softmax 层的参数。

第二个是人脸验证信号, 激励从相同个体的人脸提取的 DeepID2 特征。验证信号直接规范 DeepID2 特征并且能够有效减少个体内部的差异。常用的约束包括 L1/L2 范数和余弦相似度, DeepID2 采用如下基于 L2 范数的损失函数:

$$\text{Verif}(f_i, f_j, y_{ij}, \theta_{\text{ve}}) = \begin{cases} \frac{1}{2} \|f_i - f_j\|_2^2, & y_{ij} = 1 \\ \frac{1}{2} \max(0, m - \|f_i - f_j\|_2)^2, & y_{ij} = -1 \end{cases}$$

其中, f_i 和 f_j 是从两张对比的人脸图片中提取的 DeepID2 特征向量。 $y_{ij} = 1$ 表示 f_i 和 f_j 来自于相同个体, 在这里最小化两个 DeepID2 特征向量的 L2 距离; $y_{ij} = -1$ 表示来自于不同个体, 要求距离大于 m 。 θ_{ve} 是确认损失函数要学习的参数。

训练过程的步骤如下:

- (1) 从训练集中取两个训练样本。
- (2) 计算两张人脸图片的特征向量。
- (3) 计算参数 θ_{id} 、 θ_{ve} 、 f_i 、 f_j 、 θ_c 的梯度。
- (4) 更新 θ_{id} 、 θ_{ve} 、 θ_c 的值。
- (5) 重复步骤 1 直到收敛。

25.6.3 Fast R-CNN

Fast R-CNN^[9] 是一个用于目标检测的快速的基于区域的卷积网络, 其中也有多任务深度学习的应用。图 25-12 展示了 Fast R-CNN 的网络结构。

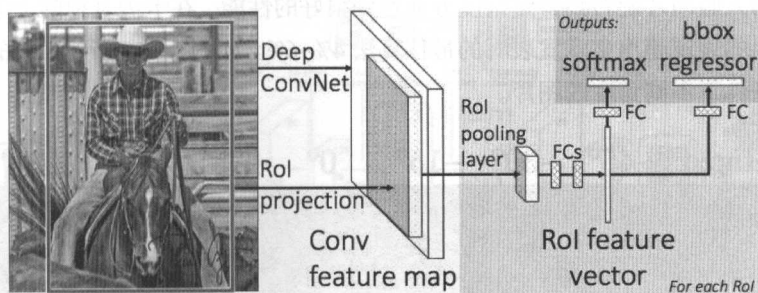


图 25-12 Fast R-CNN 网络结构示意图（将一张图片和多个 RoI（感兴趣区域）输入到一个全卷积网络，每个 RoI 会进入一个固定大小的特征图中，然后被全连接层映射到一个特征向量。每个 RoI 有两个输出：Softmax 概率和对应的矩形框回归偏移^[9]）

从图 25-12 可以看出，一个 Fast R-CNN 网络有两个输出层：第一个输出 $K+1$ 个分类的概率分布， $p = (p_0, \dots, p_k)$ ；第二个输出每个分类的矩形框回归偏移， $t_k = (t_x^k, t_y^k, t_w^k, t_h^k)$ 。每个训练 RoI 标注为一个参考标准分类 u 和一个参考标准矩形框回归对象 v ，在每个标注的 RoI 中用多任务损失函数 L 来共同训练分类和矩形框回归：

$$L(p, u, t^u, v) = L_{\text{cls}}(p, u) + \lambda [u \geq 1] L_{\text{loc}}(t^u, v)$$

其中 $L_{\text{cls}}(p, u) = -\log(p_u)$ ，是正确分类 u 的 log 损失函数。 L_{loc} 是分类 u 对应的矩形框回归对象的损失函数：

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{x, y, w, h\}} \text{smooth}_{L_1}(t_i^u - v_i)$$

其中：

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2, & \text{如果 } |x| < 1 \\ |x| - 0.5, & \text{否则} \end{cases}$$

通过 λ 来控制两个任务的损失均衡。

25.6.4 旋转人脸网络

视角和光照变化给人脸识别带来了困难，一些研究者试图通过引入姿态和光照不变特征来解决这个问题。论文 [10] 中提出了一个基于多任务学习的深度神经网络，在将一张任意

角度和光照的人脸图片旋转 to 特定角度方面达到很好的性能，在卡内基梅隆大学的 MultiPIE 人脸数据库上的表现超出文章发表时的最佳算法 4%~6%。如图 25-13 所示，新的模型可以合成正面光照条件下不同角度的图片。

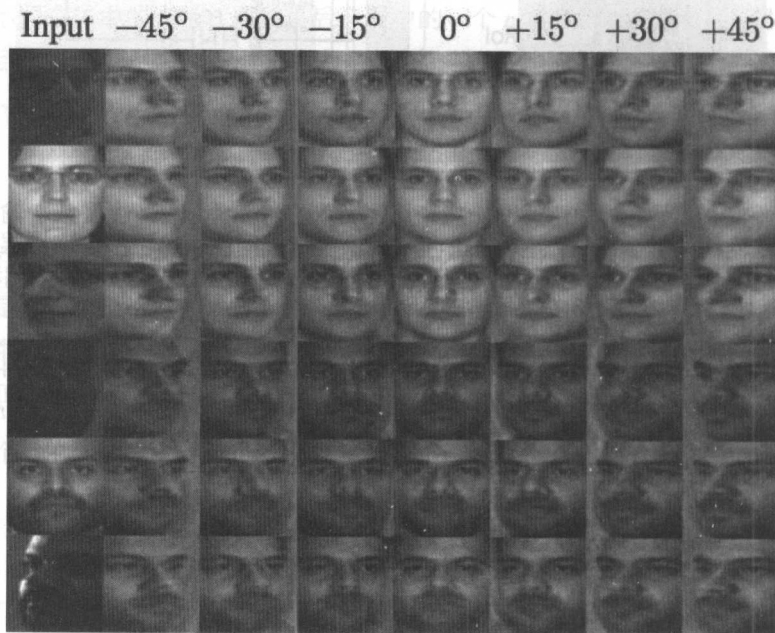


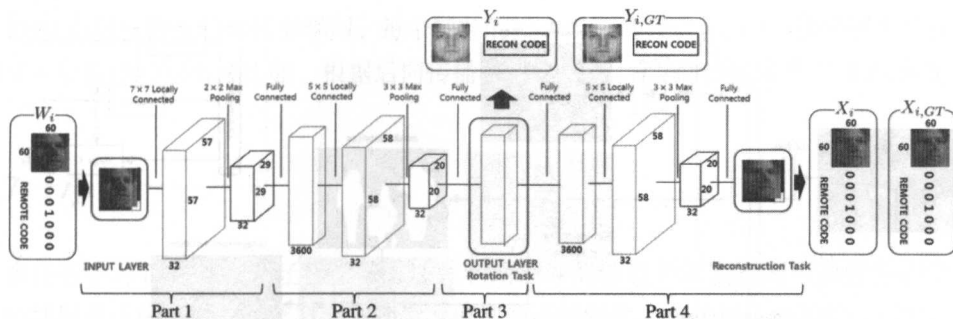
图 25-13 第一列表示 MultiPIE 数据库中两个人在不同光照条件下的输入测试图片；剩下的列是输入图片完成不同编码后的输出^[10]

早先的多任务学习模型通过共享一些层来决定公共特征，共享层后剩余的层被分割成多个任务。如图 25-13 所示，这个用于旋转人脸的深度神经网络提供了一种新颖的多任务学习方法：多任务模型共享主任务的所有层，辅助任务附加在主任务后面以提高保留个体身份的能力。

如图 25-14 所示的深度神经网络（DNN）共由 4 部分组成：抽取特征、旋转特征、生成图片和重建任务。模型的主要目标是生成特定角度的图片，但在这个任务的后面又附加一个用于重建输入图片的任务，这两个任务都使用 L2 范数作为损失函数。对于第一个任务，生成新图片的输出层的损失函数定义如下：

$$E_c = \sum_{i=1}^N |Y_{i,GT} - Y_i|_2^2$$

其中 $Y_{i,GT}$ 和 Y_i 分别是标注数据和输出数据。

图 25-14 DNN 模型结构示意图^[10]

对于第二个任务，重建输入图片的损失函数定义如下：

$$E_r = \sum_{i=1}^N |X_{i,GT} - X_i|_2^2$$

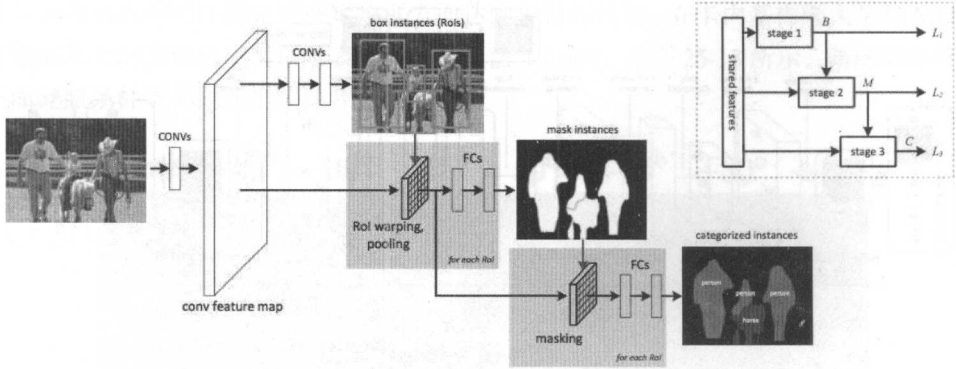
最终的损失函数是对这两个任务的损失函数的加权求和：

$$E = \lambda_c E_c + \lambda_r E_r$$

25.6.5 实例感知语义分割的 MNC

近年来，语义分割研究的进展突飞猛进，但领先的方法仍然难以准确地定位目标实例。论文 [11] 中提出了一种实现实例感知语义分割的多任务网络级联模型 (Multi-Task Network Cascades, MNC)，该模型由三个网络组成，分别对应于区分实例、估计掩码和目标分类三个子任务，这三个网络共享卷积特征并形成级联的结构。论文中在这个级联的网络结构上提出了一种实现端到端训练的算法，该算法可以扩展到有更多阶段的网络的训练。MNC 在 MS COCO 2015 分割比赛中夺得冠军，在目标检测方面超越了 Fast/Faster R-CNN。

MNC 的网络结构如图 25-15 所示。可以看出，MNC 与传统的多任务学习网络结构不同，共享卷积特征的任务并不独立。多个任务分处于不同的阶段，后一阶段的任务依赖于前一阶段的任务的输出，因此在进行后向传播时，每个任务的 loss 会影响其依赖任务的训练，从更多维度强化多个任务的训练。

图 25-15 用于实例感知语义分割的 MNC 网络结构示意图^[11]

在第一个阶段，MNC 通过 RPN（Region Proposal Network）选出对象实例的边框、预测边框的位置以及对象得分。在此阶段，MNC 并不知道对象是哪个分类。损失函数的定义如下：

$$L_1 = L_1(B(\Theta))$$

其中， Θ 表示所有需要优化的网络参数。 B 是网络的输出，表示边框列表： $B = \{B_i\}$ 和 $B_i = \{x_i, y_i, w_i, h_i, p_i\}$ ， (x_i, y_i) 是 B_i 的中心， p_i 表示边框内是对象的概率。

在第二个阶段，将共享的卷积特征和第一个阶段输出的边框作为输入，输出每个边框提名的像素级别的分割掩码。在这个阶段，掩码级别的实例依然是分类不可知的。损失函数的定义如下：

$$L_2 = L_2(M(\Theta)|B(\Theta))$$

其中， M 是这个阶段的网络输出，表示掩码列表： $M = M_i$ ， M_i 是一个 m^2 维的逻辑回归输出（ $[0, 1]$ 的连续值）。 L_2 同时依赖 M 和 B 。

在第三个阶段，将共享的卷积特征、第一个阶段输出的边框和第二个阶段输出的掩码作为输入，输出每个实例的分类得分。损失函数的定义如下：

$$L_3 = L_3(C(\Theta)|B(\Theta), M(\Theta))$$

其中， C 是这个阶段的网络输出，表示所有实例的分类预测： $C = \{C_i\}$ 。

整个级联网络的损失函数定义如下：

$$L(\Theta) = L_1(B(\Theta)) + L_2(M(\Theta)|B(\Theta)) + L_3(C(\Theta)|B(\Theta), M(\Theta))$$

$L(\Theta)$ 不同于传统的多任务学习，每个任务的损失函数的权重都是 1，这是因为后置阶段依赖于更早阶段的输出。比如，根据后向传播的链式规则， L_2 的梯度涉及了 B 的梯度。

25.7 小结

多任务学习是一种归纳迁移学习方法，利用额外的信息来源来提高当前任务的学习性能，包括提高泛化准确率、学习率和已学习模型的可理解性。在学习一个问题的同时，可以通过使用共享表示来获得其他相关问题的知识，多个任务并行训练并共享不同任务已学到的特征表示，是多任务学习的核心思想。

通过后向传播网络增加额外输出来提高神经网络的泛化能力，是一件很有意义的事情。提高泛化能力的可能原因有三：第一，不相关任务对聚合梯度的贡献相对于其他任务来说可以视为噪声，不相关任务也可以通过作为噪声源来提高泛化能力；第二，增加任务会影响网络参数的更新，比如提高了隐层有效的学习率；第三，多任务网络在所有任务之间共享网络底部的隐层，或许使用更小的容量就可以获得同水平或更好的泛化能力。因此，我们需要关注如何选择多个相关任务及数据集使网络更好地泛化。有四种机制可以帮助多任务学习网络更好地泛化：统计数据增强、属性选择、信息窃取和表示偏置。

多任务学习已经在多个领域中得到应用，本章列举了一些可应用多任务学习的场景。

- 使用未来预测现在。
- 多种表示和度量。
- 时间序列预测。
- 使用不可操作特征。
- 使用额外任务来聚焦。
- 有序迁移。
- 多个任务自然地出现。
- 将输入变成输出。

深度学习网络是具有多个隐层的神经网络，在深度学习网络中应用多任务学习是一种很自然的想法。在机器视觉领域多任务学习被广泛应用，主要方式包括：多个任务并行输出，同时做分类和回归或使用不同的损失函数；多个任务如流水线般，辅助任务附加在主任务后面；多个任务存在依赖关系，形成级联的网络结构。常见的多任务学习网络结构如图 25-16 所示。

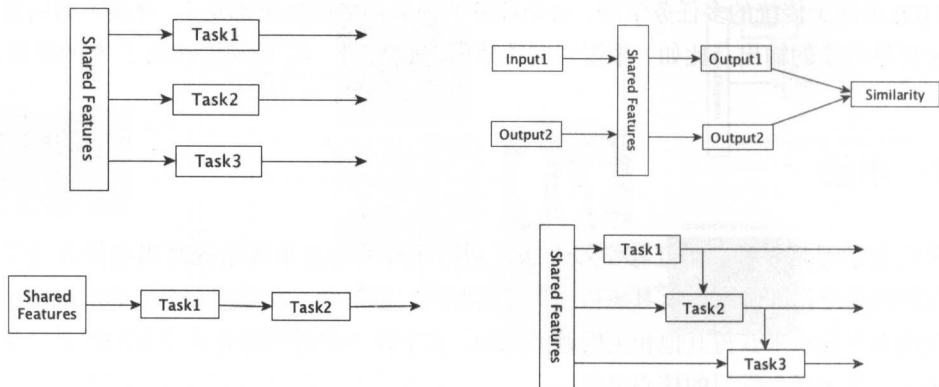


图 25-16 4 种常见的多任务学习网络结构示意图

多任务学习将会在越来越多的领域作为一种提高神经网络学习能力的手段被广泛应用。

参考文献

- [1] R. Caruana. Multitask Learning. Mach Learn, vol. 28, no. 1, pp. 41-75, Jul. 1997.
- [2] Multi-task learning. [Online]. Available. https://en.wikipedia.org/wiki/Multi-task_learning.
- [3] L. Pratt and S. Thrun. Guest Editors' Introduction. Mach Learn, vol. 28, no. 1, pp. 5-5, Jul. 1997.
- [4] T. G. Dietterich, L. Pratt, and S. Thrun, Eds.. Machine Learning-Special issue on inductive transfer. Mach Learn, vol. 28, no. 1, 1997.
- [5] L. Holmstrom and P. Koistinen. Using additive noise in back-propagation training. IEEE Trans. Neural Netw., vol. 3, no. 1, pp. 24-38, Jan. 1992.
- [6] Z. Zhang, P. Luo, C. C. Loy, and X. Tang. Facial Landmark Detection by Deep Multi-task Learning. in Computer Vision-ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VI, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 94-108.
- [7] Y. Sun, Y. Chen, X. Wang, and X. Tang. Deep Learning Face Representation by Joint Identification-Verification. in Advances in Neural Information Processing Systems 27, Z. Ghahra-

mani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 1988-1996.

[8] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, 2007.

[9] R. Girshick. Fast R-CNN. in The IEEE International Conference on Computer Vision (ICCV), 2015.

[10] J. Yim, H. Jung, B. Yoo, C. Choi, D. Park, and J. Kim. Rotating Your Face Using Multi-Task Deep Neural Network. in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[11] J. Dai, K. He, and J. Sun. Instance-Aware Semantic Segmentation via Multi-Task Network Cascades. in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

26

模型压缩

随着移动互联网的突飞猛进以及智能手机硬件的不断升级，越来越多的功能从云端（或服务器端）转到手机客户端来完成，从而实现无须联网且更快速地完成相关功能。深度学习的图像识别、语音识别及自然语言处理等功能也开始客户端化，而面临的第一个问题就是深度学习模型较大而智能手机内存还不足的冲突，模型压缩是这个问题的解决方法之一。本章将带大家一起来了解最新的模型压缩研究成果。

26.1 模型压缩的必要性

在深度学习向我们展示其强大的功能时，我们也需要注意到，深度学习用到了大量的参数。让我们来计算一下著名的 AlexNet 的模型有多大，图 26-1 显示了 AlexNet 的网络结构^[1]。AlexNet 由 5 个卷积层和 3 个全连接层组成。

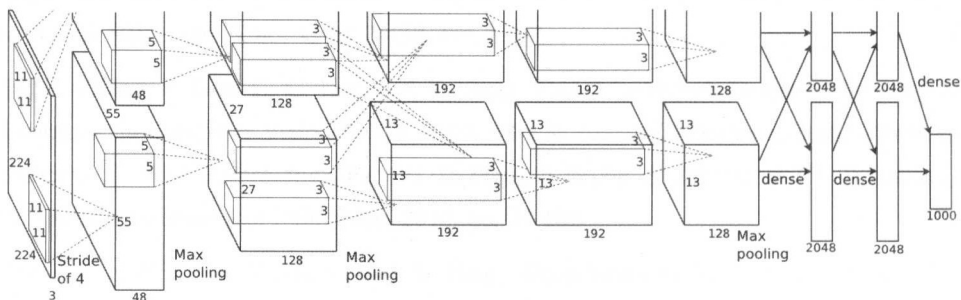


图 26-1 AlexNet 网络结构示意图^[1]

由表 26-1 我们不难算出, AlexNet 总计使用了 5800 多万个参数。如果每个参数都以单精度浮点数在计算机中存储和表示的话, 那么每个参数占用的存储空间为 4B, 5800 多万个参数总共要使用 200MB 以上的存储空间。

表 26-1 AlexNet 层的参数结构

层号	输入	Kernel	Num_output	输出	参数个数
1	$227 \times 227 \times 3$	11	96	$55 \times 55 \times 96$	$96 \times (3 \times 11 \times 11 + 1) = 34944$
2	$27 \times 27 \times 96$	5	256	$27 \times 27 \times 256$	$256 \times (96 \times 5 \times 5 + 1) = 614656$
3	$13 \times 13 \times 256$	3	384	$13 \times 13 \times 384$	$384 \times (256 \times 3 \times 3 + 1) = 885120$
4	$13 \times 13 \times 384$	3	384	$13 \times 13 \times 384$	$384 \times (384 \times 3 \times 3 + 1) = 1327488$
5	$13 \times 13 \times 384$	3	256	$13 \times 13 \times 384$	$256 \times (384 \times 3 \times 3 + 1) = 884992$
6	$6 \times 6 \times 256$	—	—	4096	$(256 \times 6 \times 6 + 1) \times 4096 = 37752832$
7	4096	—	—	4096	$(4096 + 1) \times 4096 = 16781312$
8	4096	—	—	1000	$(4096 + 1) \times 1000 = 4097000$

如此巨大的模型只适用于云端。但是通过云端来处理数据, 其隐私性较差, 传输数据也要占用网络带宽和时间。如果可以在手持设备上使用 DNN 模型, 其优点不言自明——更好的隐私保障, 不需要占用网络带宽, 没有网络传输数据的时延。但如此巨大的模型放在手持设备上不够现实。首先, 较大的应用在下载时会受到阻拦, 例如苹果的应用商店只允许通过 WiFi 来下载超过 100MB 的应用。其次, 巨大的模型占用手持设备宝贵的存储空间, 应用在加载时占用如此巨大的内存显然是不现实的。第三, 大模型的计算量很大, 能量消耗很大。巨大的神经网络在运行时需要很多次内存访问和大量的点乘计算, 而在使用电池的手持设备上使用如此耗能的应用也是非常不现实的。能耗问题主要来自于对内存的访问。在 45nm 的 CMOS 工艺下, 32 位的浮点加只耗能 0.9pJ, 32 位的 SRAM Cache 存取需要耗能 5pJ, 而 32 位的 DRAM 存取竟然要耗能 640pJ。对于有 10 亿个连接的神经网络来说, 如果每秒处理 20 帧数据, 仅 DRAM 存取一项, 就需要 $(20\text{Hz})(1\text{G})(640\text{pJ}) = 12.8\text{W}^{[2]}$ 。

那么真的需要这么多参数才能够完成相应的任务吗? 如果使用更小的模型, 能否达到相同或者相似的水平呢? 如果真的存在更小的模型, 要怎么做才能找到它呢? 读完本章内容, 读者就会得到答案了。

26.2 较浅的网络

从直观上想，使用较浅的网络可以缩小网络模型的大小。然而，已经有实验显示，浅层网络在表达能力上无法与深度模型相提并论^[3]。L.J. Ba 等人尝试采用模仿深度网络的方法使用大量的训练数据构造了浅层网络，虽然在准确率上与深度网络近似，但是使用了更多的模型参数^[4]。从这点上讲，使用较浅的网络无法真正达到压缩模型的效果。参考文献 [5] 从理论上证明了网络的表达能力随着深度的增加呈指数级增长，浅层网络无法高效地模拟深度模型。

26.3 剪枝

一般来说，当训练数据固定时，具有太多参数的网络的泛化能力不好。然而，参数太少的网络又无法精确地表示数据。最优的泛化能力是在训练错误与网络复杂性之间寻找平衡。

在网络模型中冗余连接带来的信息量几乎为 0。把已经训练好的模型中的冗余连接删掉，也可以缩小网络模型的大小。早在 1989 年，Pratt L.Y. 就提出了可以使用 Weight Decay 的方法来获得更多的值为 0 的参数^[4]。参考文献 [6] 指出，通过去掉那些不重要的参数，可以使模型具有更好的泛化能力，需要更少的训练数据，提高了训练/测试的速度。最近，Han 等人通过研究参数与连接的关系，用剪枝的方法大幅地减小了网络模型的大小^{[3][5]}。

剪枝也是具有仿生学意义的。参考文献 [7] 指出，人类大脑在发育过程中，突触连接数量会降低，如表 26-2 所示。人类大脑的突触在刚出生时有 50 万亿个。1 岁时，达到顶峰，高达 1000 万亿个。此时，剪枝开始。10 岁大小的孩子只有 500 万亿个突触。“剪枝”机制的含义是把冗余连接去掉。

表 26-2 人类大脑的突触剪枝机制

年龄	连接数量（个）	状态
出生时	50 万亿	初形成
1 岁	1000 万亿	峰值
10 岁	500 万亿	剪枝并稳定

参考文献 [6] 中提出了 Optimal Brain Damage（OBD）的剪枝方法，并从理论上证明了其正确性。删掉参数的本质是：删掉不重要的参数。也就是说，删掉这个参数对训练目标函数的影响最小。OBD 使用二阶偏导数来计算“显著性”，那么该如何定义显著性呢？最简单、

直观的方法是：删掉每个参数，重新计算目标函数，由目标函数的差值来定义显著性。但这种方法的计算量太大，不可行。幸运的是，有一种方法可以从理论上分析近似参数向量的改变导致的目标函数的变化。使用泰勒展开式来计算目标函数 E ，那么目标函数的变化可以写成：

$$\delta E = \sum_i g_i \delta u_i + \frac{1}{2} \sum_i h_{ii} \delta u_i^2 + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta u_i \delta u_j + O(\|\delta U\|^3) \quad (26.1)$$

其中 U 表示参数， δU 是 U 的变化， δu_i 是 δU 的元素， G 是 E 对 U 的偏导数， g_i 是 G 中的元素。 h_{ij} 是海森矩阵（Hessian Matrix）的元素。有：

$$g_i = \frac{\partial E}{\partial u_i} \quad (26.2)$$

$$h_{ij} = \frac{\partial^2 E}{\partial u_i \partial u_j} \quad (26.3)$$

剪枝的目标就是寻找这样一组参数，当删掉这些参数时， E 的增长最少。在通常情况下，式 (26.1) 的计算量太大了。海森矩阵 H 的元素个数是参数个数的平方，而且直接求解海森矩阵也很困难。参考文献 [6] 中做出这样一个假设：由于删掉一些参数而导致的目标函数的变化 δE 等于删掉每一个参数引起的 E 的变化之和，交叉项被忽略。换句话说，这个假设认为海森矩阵都是对角阵，非对角线上的元素都是 0。根据这个假设，式 (26.1) 中的第三项可以被忽略。由于此时在 E 的一个局部最优点，所以第一项也可以被忽略。还有，在局部极小点，所有的 h_{ij} 都非负，所以参数的任何变化都会导致 E 的增长。因此，式 (26.1) 可以近似写为：

$$\delta E = \frac{1}{2} \sum_i h_{ii} \delta u_i^2 \quad (26.4)$$

于是，这个问题就转化为如何计算二阶导数 h_{ii} 。这里假设损失函数使用的是均方误差（MSE）。假设 X_i 为结点 i 的输出， a_i 是结点 i 的输入加权和， f 是激活函数， w_{ij} 表示从结点 j 到结点 i 的权重。

$$x_i = f(a_i) \quad (26.5)$$

$$a_i = \sum_j w_{ij} x_j \quad (26.6)$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_{ij}} = \frac{\partial E}{\partial a_i} x_j \quad (26.7)$$

$$\frac{\partial^2 E}{\partial w_{ij}^2} = \frac{\partial \frac{\partial E}{\partial w_{ij}}}{\partial w_{ij}} = \frac{\partial \frac{\partial E}{\partial a_i}}{\partial w_{ij}} x_j = \frac{\partial^2 E}{\partial a_i^2} x_j^2 \quad (26.8)$$

$$\frac{\partial E}{\partial x_i} = \sum_l w_{li} \frac{\partial E}{\partial a_l} \quad (26.9)$$

$$\frac{\partial^2 E}{\partial x_i^2} = \sum_l w_{li} \frac{\partial^2 E}{\partial a_l^2} \frac{\partial a_l}{\partial x_i} = \sum_l w_{li}^2 \frac{\partial^2 E}{\partial a_l^2} \quad (26.10)$$

$$\frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial x_i} \frac{\partial x_i}{\partial a_i} = f'(a_i) \frac{\partial E}{\partial x_i} \quad (26.11)$$

由式 (26.9) 和式 (26.11) 可以推导得到:

$$\frac{\partial^2 E}{\partial a_i^2} = f'(a_i) \frac{\partial \frac{\partial E}{\partial x_i}}{\partial a_i} + f''(a_i) \frac{\partial E}{\partial x_i} = f'(a_i)^2 \sum_l w_{li}^2 \frac{\partial^2 E}{\partial a_l^2} + f''(a_i) \frac{\partial E}{\partial x_i} \quad (26.12)$$

$$E = \sum_i (d_i - x_i)^2 \quad (26.13)$$

对于输出层, 有:

$$\frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial x_i} \frac{\partial x_i}{\partial a_i} = -2(d_i - x_i) f'(a_i) \quad (26.14)$$

$$\frac{\partial^2 E}{\partial a_i^2} = 2f'(a_i)^2 - 2(d_i - x_i) f''(a_i) \quad (26.15)$$

OBD 中提出的算法如下:

- (1) 选择合适的网络结构。
- (2) 训练网络, 获得合适的结果。
- (3) 根据式 (26.8) 为每个参数计算其二阶偏导。
- (4) 根据式 (26.4) 为每个参数计算其显著值。
- (5) 按照显著值给参数排序, 删除显著值低的参数。
- (6) 返回第 2 步迭代。

OBS^[8] 与 OBD 相比, 去掉了海森矩阵都是对角阵的假设, 提出了用迭代的方式计算海森矩阵, 感兴趣的读者可以自行阅读。

在参考文献 [11] 中, 为了避免计算二阶偏导数, 假设绝对值较小的参数就是不重要的参数。这一假设在如今的大规模网络中获得了良好的剪枝效果。其流程如图 26-2 所示。

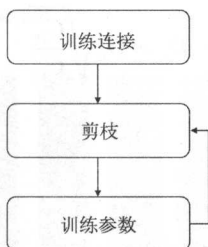


图 26-2 流水训练三步走

与传统的训练不同的是，这里的第一步并不是为了得到这些参数的最终值，而是为了得到哪些参数更重要。在第二步中，绝对值较小的连接被剪掉。如图 26-3 所示，在压缩连接时，那些没有输入或者没有输出的神经元自然可以被压缩，这些神经元的相关连接又可以被删除，稠密网络变为稀疏网络。第三步，在剩下的网络结构中，重新训练各个参数的取值。

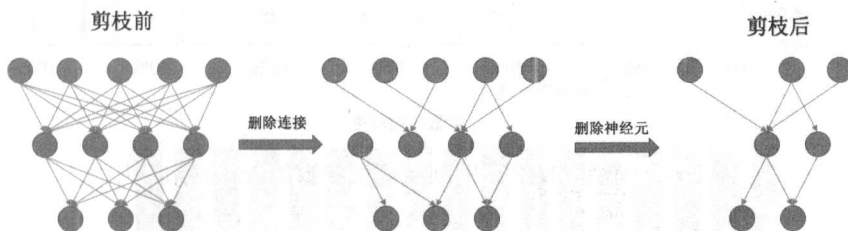


图 26-3 剪枝前后的神经元及连接

在剪枝过程中使用的正则方法会影响到剪枝和再训练的精度。L1 正则会对非 0 的参数进行惩罚，所以会产生较多的 0 附近的参数。如果使用 L1 正则，在剪枝之后能获得较好的精确率，但是在重新训练后，其精度不如使用 L2 正则。参考文献 [11] 中提到迭代使用 L2 正则做剪枝和重新训练，在相同的压缩率下会得到最好的精度，如图 26-4 所示。

在深度神经网络中通常会使用 Dropout 来避免过拟合。在再训练阶段，仍然会使用 Dropout 技术。使用 Dropout 时，在训练阶段，每个参数被以一定的概率值丢弃；但是在测试阶段，所有的参数都会参与计算。剪枝与 Dropout 的区别在于，被剪掉的参数永远不会再回来参与计算。剪枝后，由于参数稀疏化，过拟合的程度被减轻，所以在再训练阶段，Dropout 的比例也应该降低。

令 C_i 是第 i 层的连接数， C_{i0} 是原网络的连接数， C_{ir} 是再训练后的连接数， N_i 是第 i 层的神经元数量。于是有：

$$C_i = N_i N_{i-1} \quad (26.16)$$

$$D_r = D_o \sqrt{\frac{C_{ir}}{C_{io}}} \quad (26.17)$$

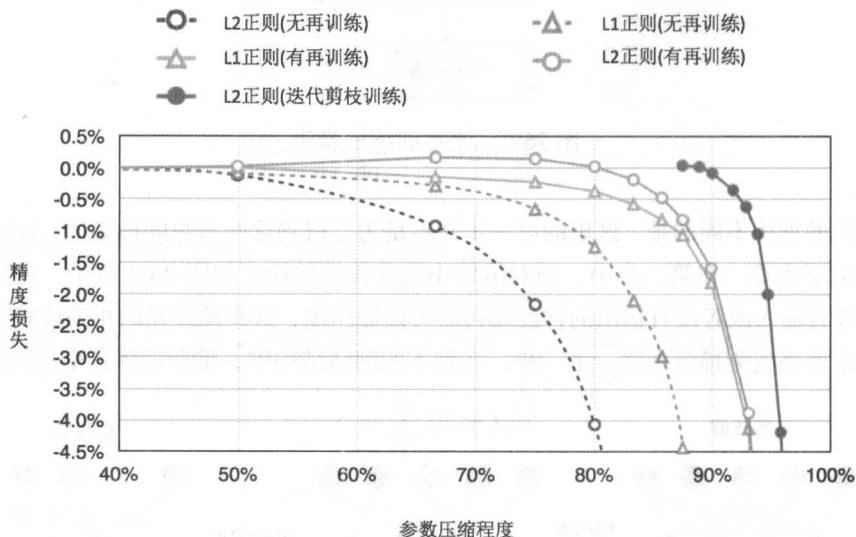


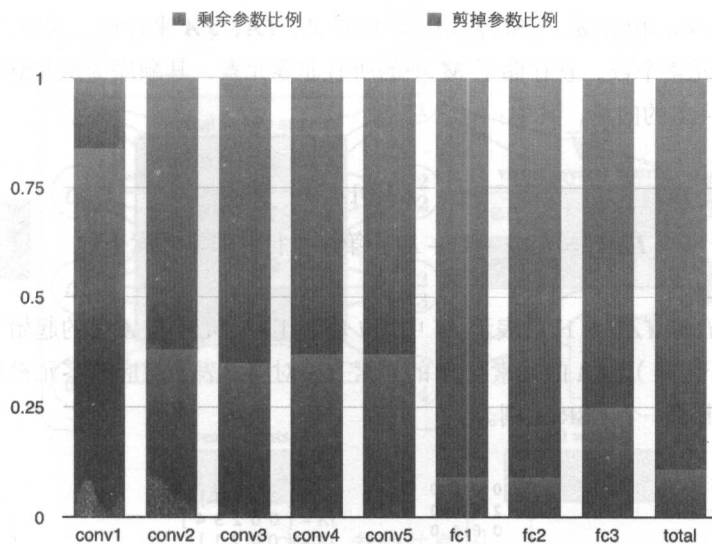
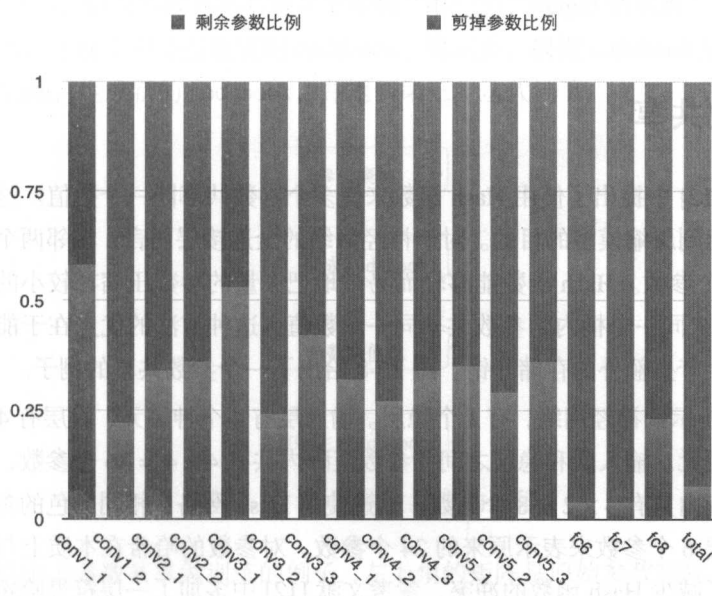
图 26-4 使用 L1/L2 正则削减模型参数的 Top 5 精度^[5]

由于 Dropout 作用于神经元，所以 C_i 与 N_i 是平方关系。根据式 (26.16)，可以得到再训练网络的 Dropout 比例 D_r 与原始网络的 Dropout 比例 D_o 的关系式 (26.17)。

剪枝之后，如果把剩下的神经元重新初始化再重新训练，效果会比较差。所以，在重新训练被剪枝的层时，需要保留剩下的这些参数。在重新训练全连接层时，需要固定卷积层；在重新训练卷积层时，需要固定全连接层^[9]。

剪枝可以循序渐进，一次剪枝、再训练的过程完成之后，可以继续迭代这个过程。采用迭代的方法，比只使用一次剪枝过程能够取得更好的压缩效果。参考文献 [11] 中指出，对 AlexNet 做压缩时，在保证精度的前提下，一次压缩只能压缩到 5x；而采用迭代的方法，能够压缩到 9x。

图 26-5 和图 26-6 分别展示了剪枝方法作用于 AlexNet^[1] 和 VGGNet^[2] 的效果。不难看出，卷积层对剪枝比较敏感，所以在剪枝过程中需要对卷积层做较多的保留；而全连接层则对剪枝不太敏感，所以可以对全连接层做比较多的压缩。从 26.1 节关于 AlexNet 各层参数的计算可以看出，全连接层的参数较多，所以对全连接层做较大比例的压缩能够对整个模型起到关键的压缩作用。参考文献 [10] 中用类似的方法对 LSTM 做了剪枝，能够压缩掉 90% 的连接。

图 26-5 AlexNet 的剪枝效果^[11]图 26-6 VGGNet 的剪枝效果^[11]

我们已经介绍了如何把稠密矩阵剪枝形成稀疏矩阵，那么如何存储稀疏矩阵呢？通常采用 Compressed Sparse Row (CSR) 或者 Compressed Sparse Column (CSC) 来存储稀疏矩阵。不失一般性，我们以 CSR 为例来介绍稀疏矩阵的存储。

对于稀疏 $m \times n$ 矩阵 M , CSR 使用 3 个向量 A 、 IA 、 JA 来存储。其中, A 的长度为矩阵 M 中的非零元素个数, 它存储了 M 中的所有非零元素, 其顺序为从左到右、从上至下。 IA 是长度为 $m+1$ 的向量, 其中:

$$IA[0] = 0 \quad (26.18)$$

$$IA[i] = IA[i-1] + M \text{ 中第 } i-1 \text{ 行的非零元素个数} \quad (26.19)$$

于是, $IA[i]$ 和 $IA[i+1]$ 就表示 M 中第 i ($i \geq 1$) 行元素在 A 中的起始下标 (闭区间) 和终止下标 (开区间)。 JA 的元素与 A 的元素一一对应, 表示对应非零元素在 M 中的列下标。图 26-7 给出了一个 CSR 示例。

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix} \quad \begin{aligned} A &= [2 \ 4 \ 1 \ 3] \\ IA &= [0 \ 0 \ 2 \ 3 \ 4] \\ JA &= [0 \ 1 \ 2 \ 1] \end{aligned}$$

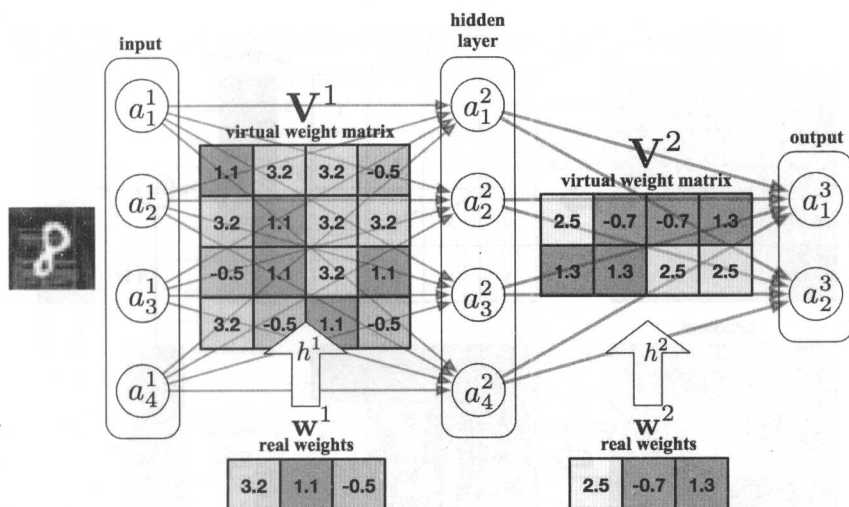
图 26-7 稀疏矩阵的 CSR 表示

26.4 参数共享

参考文献 [12] 中提出了使用 Hash 函数来使多个参数共享同一个数值。这种方法通过限制桶的个数来达到压缩模型的目的。对于神经网络的全连接层而言, 相邻两个全连接层需要 $(n_l + 1) * n_{l+1}$ 个参数。Hash 函数能够轻而易举地把大量的参数压缩在较小的范围内, 所有被 Hash 函数落入同一个桶内的参数共享同一个数值。这种方法的优点在于能够随意调节压缩比, 而且不会产生额外的存储开销。图 26-8 给出了一个参数共享的例子。

如图 26-8 所示的神经网络, 有 1 个隐层。输入层有 4 个神经元, 隐层有 4 个神经元, 输出层有 2 个神经元。输入层和隐层之间的参数矩阵内共有 $4 \times 4 = 16$ 个参数, 隐层和输出层之间的参数矩阵内共有 $4 \times 2 = 8$ 个参数。通过使用 Hash 网络, 相同颜色的参数对应同一个哈希桶, 只需要 6 个参数来表示原来的 24 个参数。对参数的哈希在本质上与对特征的哈希是一致的。为了减少 Hash 函数的冲突, 参考文献 [12] 中多加了一层符号哈希, 给参数乘上 +1 或 -1。

与参考文献 [12] 不同, 参考文献 [2] 中提出多个数值上相近的参数组成一簇, 共同使用一个参数, 在压缩 AlexNet 时, 卷积层使用 256 个不同的数值, 全连接层使用 32 个不同的数值。这样就不再需要使用 32 位浮点数来表示每个参数了。而是构建一个 codebook 来记录所有的参数数值, 然后用记录在 codebook 中的下标来表示各个参数。这样就可以使用 8/5 比特来记录每个参数了。

图 26-8 参数共享^[12]

如图 26-9 所示, 相近参数共享共有 4 个步骤。第一步, weight 做聚类, 一般采用 k 近邻做聚类。第二步, 生成由中心点组成的 codebook。第三步, 根据 codebook 把各参数数值化。第四步, 重新训练, 生成新的 codebook, 并返回第三步迭代训练。

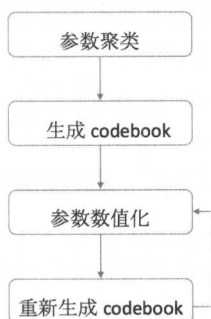
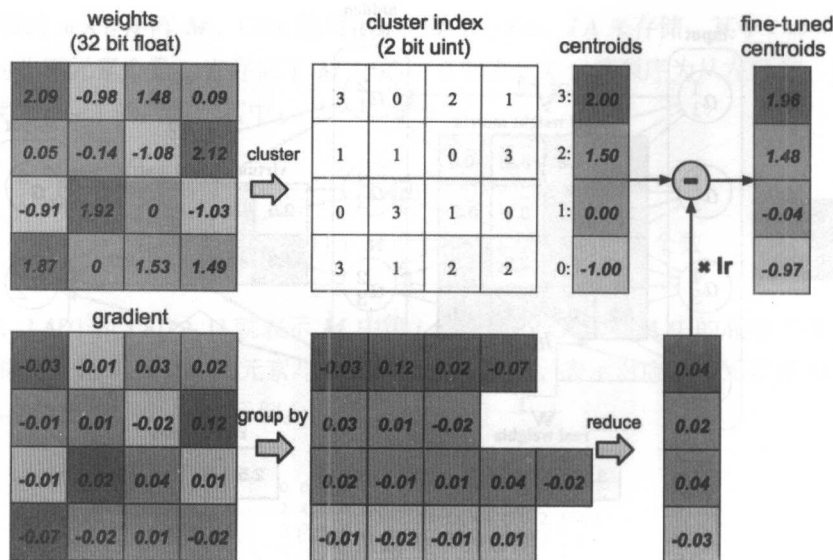


图 26-9 相近参数共享的过程

图 26-10 给出了参数共享做训练的例子。左上角的矩阵是原始参数矩阵, 每个参数由 32 位浮点数来表示。对原始参数矩阵做聚类, 划分为 4 类, 每类的中心点由该类内的平均值来表示, 构成了 codebook。这时, 原始矩阵就可以改为存储聚类的下标。由于这个例子中只有 4 类, 所以用 2 比特就能表示每个参数。原始梯度按照同样的分组方式分为 4 组, 为每组的梯度求和来代表整组参数的梯度。然后乘上学习率 (Learning Rate), 再被 codebook 中的原始参数值减去, 就得到了更新后的参数。

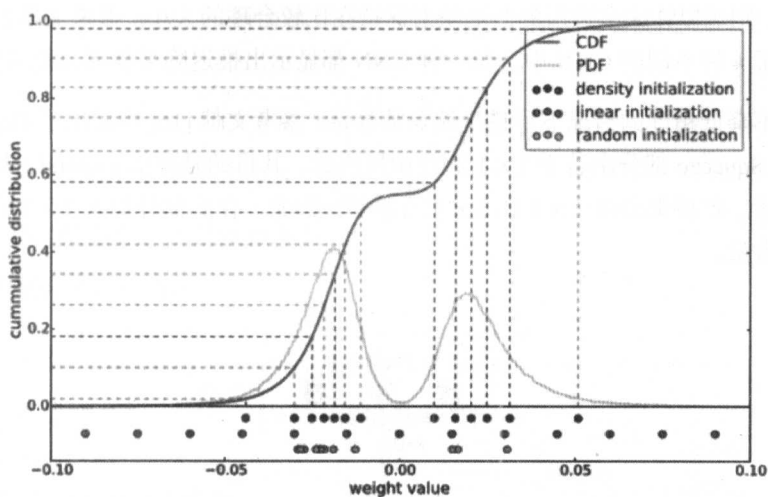
图 26-10 参数共享的梯度下降法^[2]

假设在参数共享过程中共有 k 个簇，那么需要 $\log_2 k$ 比特来表示每个索引。假设在原网络中每个参数由 b 比特来表示，如果有 n 个参数，那么在原网络中需要 $n * b$ 比特来表示。参数共享后，需要 $n \log_2 k + kb$ 比特来表示所有的参数和 codebook。于是，压缩率可以计算如下：

$$r = \frac{nb}{n \log_2(k) + kb} \quad (26.20)$$

参考文献 [2] 中提到，使用参数共享后，卷积层使用 8 比特来表示，全连接层使用 5 比特来表示，模型的准确率没有下降。

在 26.3 节中，我们提到在模型中绝对值较大的参数比绝对值较小的参数更重要。那么如何选择聚类的初始值呢？图 26-11 给出了三种不同的选择初始值的方法。图中的 CDF 表示累积密度函数，PDF 表示概率密度函数。如果采用完全随机的方法来选择初始值或者基于密度来选择初始值，则会使初始值集中在概率密度大的参数附近。绝对值较大的参数出现的概率低，采用这种方法就没有办法很好地表征这类参数了。如果采用线性的方法来选择初始值，则对绝对值较大的参数的表征会更友好。实验证明，在这三种方法中，线性的方法在同样的压缩率下准确率最高^[2]。

图 26-11 三种选择初始值的方法^[2]

26.5 紧凑网络

在神经网络的每一层使用更紧凑的结构同样可以节约存储和计算资源。比如, 在 NIN^[13], GoogLeNet^[14]、Residual-Net^[15] 中, 采用全局平均池化层 (Global Average Pooling) 来代替全连接层, 获得了很好的结果。SqueezeNet^[16] 采用 1×1 的卷积核代替部分 3×3 的卷积核用来构建非常紧凑的神经网络, 参数数量能减少 50 倍。

SqueezeNet 提出了从结构上减少参数的方法:

- (1) 使用较小的卷积核。例如, 使用 1×1 的卷积核代替部分 3×3 的卷积核可以使参数数量减少 9 倍。
- (2) 降低较大卷积核的输入 Channel (通道)。由于某个卷积层的参数总数是:

$$\text{输入 Channel 数量} \times \text{卷积核数量} \times \text{卷积核大小} \quad (26.21)$$

所以降低输入 Channel 的数量也可以减少参数的总数。

然而, 一味地降低参数数量, 会降低模型对数据的表达能力。所以, 推迟降采样的时机则能够保证模型在计算过程中拥有较大的激活映射 (Activation Map)。也就是说, 参数虽然减少了, 但计算量没有减少, 模型的表征能力没有变差。如果在比较靠前的卷积层使用了比较大的降采样间隔, 那么大多数层的激活映射就比较小。如果推迟降采样, 或者缩小降采样

间隔的大小,则可以让大多数层的激活映射保持在比较合理的大小。参考文献[17]中把推迟降采样用在了4种不同的CNN中,每一种CNN都显示出推迟降采样可以提高模型的精度。

为了在不降低精度的情况下尽量压缩参数数量,参考文献[16]中提出了Fire结构,如图26-12所示。Squeeze部分由若干 1×1 的卷积核构成,其目的是降低Expand(扩展)部分的输入通道数量。扩展部分由 1×1 和 3×3 的卷积核构成,在此部分加入 1×1 的卷积核是为了减少参数数量。

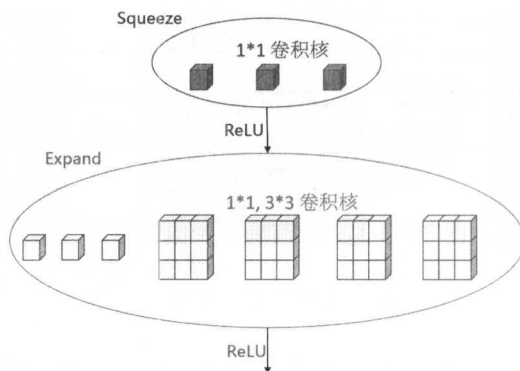


图 26-12 Fire 结构

Squeeze部分的 1×1 卷积核数量用 $s1 \times 1$ 表示,Expand部分的 1×1 卷积核数量用 $e1 \times 1$ 表示, 3×3 卷积核数量用 $e3 \times 3$ 表示。使用Fire结构时, $s1 \times 1$ 要小于上一层的Fire的 $(e1 \times 1 + e3 \times 3)$ 才能起到减少参数的作用。

令人欣喜的是,紧凑网络与前面所提的剪枝、参数共享的方法可以同时使用。也就是说,使用紧凑的网络结构训练好之后,还可以使用剪枝和参数共享来进一步压缩模型的参数数量和模型大小。在参考文献[16]中使用紧凑网络将参数数量减少到1/50后,又使用剪枝和参数共享进一步将模型参数数量减少到1/510。

26.6 二值网络

最近,DNN开始采用二值网络。BinaryConnect提出了在传播时使用+1或-1来代替原来的参数,那么原来网络中的乘加操作就转化成了简单的加减操作^[18]。这不仅降低了模型的存储空间,也提高了模型的运算速度。那么如何将实数域上的参数映射到二值上呢?参考

文献 [18] 中给出了两种方法。一种方法是确定性的方法，由参数的符号来决定：

$$w_b = \begin{cases} +1, & w \geq 0 \\ -1, & \text{其他} \end{cases} \quad (26.22)$$

其中 w_b 是二值参数， w 是实数参数。

另一种方法是随机的方法：

$$w_b = \begin{cases} +1, & \text{传播概率为 } p = \sigma(w) \\ -1, & \text{传播概率为 } 1 - p \end{cases} \quad (26.23)$$

其中 σ 是 Hard Sigmoid 函数：

$$\sigma(x) = \text{clip}\left(\frac{x+1}{2}, 0, 1\right) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right) \quad (26.24)$$

DNN 的训练过程分为三步。

- (1) 给定 DNN 的输入，逐层前向计算，直到算出顶层的结果。这一步被称为前向传播。
- (2) 从 DNN 的顶层开始逐层计算训练目标的偏导数，直到第一个隐层。这一步被称为后向传播。
- (3) 计算对每一层参数的偏导数，利用旧值和偏导数来更新参数。这一步被称为参数更新。

在 BinaryConnect 中，只有第 1 步和第 2 步是在二值网络上完成的，而参数更新（第 3 步）是在实数网络上完成的。这是因为为了使 SGD 更好地工作，必须在参数更新时保证精度。算法 26-1 给出了 BinaryConnect 的 SGD 训练过程。

算法 26-1 BinaryConnect 的 SGD 训练过程

C : Mini-Batch 的损失函数， L : 层数， η : 学习率

前向传播：

将 w_{l-1} 二值化，得到 w_b

对于所有的 k (从 1 到 L)，根据 a_{k-1}, w_b, b_{l-1} 计算 a_k

后向传播：

初始化输出层偏导数 $\frac{\partial C}{\partial a_L}$

从层 L 到层 2, 根据 $\frac{\partial C}{\partial a_k}, w_b$ 计算 $\frac{\partial C}{\partial a_{k-1}}$
参数更新:

根据 $\frac{\partial C}{\partial a_k}$ 和 a_{k-1} 计算 $\frac{\partial C}{\partial w_b}$ 和 $\frac{\partial C}{\partial b_{t-1}}$

$$w_t = \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$$

$$b_t = b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$$

在 BinaryNet^[19] 中不仅使用了二值的参数, 而且使用了二值的激活函数。在 XNOR-Net^[20] 中延伸了 BinaryConnect 和 BinaryNet 的方法, 但是使用了不同的计算方式来近似网络。

与 BinaryConnect 对应, 参考文献 [20] 中提出了二值参数网络使用 $\{+1, -1\}$ 来近似实数参数网络。

用 \mathbf{W} 表示实数参数矩阵, \mathbf{B} 表示二值参数矩阵, \mathbf{I} 表示输入, 正实数 α 表示缩放因子。所以二值参数网络的目标是求出这样的 \mathbf{B} , 使得:

$$\mathbf{W} \approx \alpha \mathbf{B} \quad (26.25)$$

于是卷积操作就可以近似为:

$$\mathbf{I} * \mathbf{W} \approx (\mathbf{I} \oplus \mathbf{B}) \alpha \quad (26.26)$$

其中, $*$ 表示卷积操作, \oplus 表示没有乘法的卷积操作。这是因为 \mathbf{B} 由 $\{+1, -1\}$ 组成, 所有的卷积操作都可以由加减来表示。为了寻找式 (26.25) 的最优解, 问题可以转化为求如下最优问题:

$$J(\mathbf{B}, \alpha) = \|\mathbf{W} - \alpha \mathbf{B}\|^2 \quad (26.27)$$

$$\alpha^*, \mathbf{B}^* = \arg \min_{\alpha, \mathbf{B}} J(\mathbf{B}, \alpha) \quad (26.28)$$

展开式 (26.27), 可以得到:

$$J(\mathbf{B}, \alpha) = \alpha^2 \mathbf{B}^T \mathbf{B} - 2\alpha \mathbf{W}^T \mathbf{B} + \mathbf{W}^T \mathbf{W} \quad (26.29)$$

由于 \mathbf{B} 内元素取值 $(-1, 1)$, \mathbf{W} 是已知变量, 所以上式可重写为:

$$J(\mathbf{B}, \alpha) = \alpha^2 n - 2\alpha \mathbf{W}^T \mathbf{B} + c \quad (26.30)$$

求 J 的最小值可以转化为求上式第二项的最大值，于是问题转化为：

$$\mathbf{B}^* = \arg \max_{\mathbf{B}} \{\mathbf{W}^T \mathbf{B}\} \quad s.t. \quad \mathbf{B} \in \{+1, -1\}^n \quad (26.31)$$

当 $W_i \geq 0$ 时, B_i 取 +1; 当 $W_i < 0$ 时, B_i 取 -1, 就得到了最优解。所以其最优值是：

$$\mathbf{B}^* = \text{sign}(\mathbf{W}) \quad (26.32)$$

根据式 (26.30) 求 J 对 α 的偏导, 令其等于 0, 可以得到 α 的最优解 α^* ：

$$\alpha^* = \frac{\mathbf{W}^T \mathbf{B}^*}{n} = \frac{\mathbf{W}^T \text{sign}(\mathbf{W})}{n} = \frac{\sum [W_i]}{n} = \frac{1}{n} \|\mathbf{W}\|_{l_1} \quad (26.33)$$

二值参数网络的参数训练过程与 BinaryConnect 相似, 在前向传播和后向传播中使用了二值, 参数更新时使用了实数参数。

在 XNOR-Net 中, 特征映射也用二值来表示, 于是二值参数网络中的加减操作就转化成了二值卷积操作, 也就是位操作。那么该如何寻找特征映射的近似值呢? 该问题就是求解式 (26.34) 的最优解。

$$\mathbf{X}^T \mathbf{W} \approx \beta \mathbf{H}^T \alpha \mathbf{B}, \text{ 其中 } \mathbf{H}, \mathbf{B} \in \{-1, +1\}^n, \beta, \alpha \in \mathbb{R}^+ \quad (26.34)$$

$$\alpha^*, \mathbf{B}^*, \beta^*, \mathbf{H}^* = \arg \min_{\alpha, \mathbf{B}, \beta, \mathbf{H}} \|\mathbf{X} \odot \mathbf{W} - \beta \alpha \mathbf{H} \odot \mathbf{B}\| \quad (26.35)$$

定义 $\mathbf{Y}_i = \mathbf{X}_i \mathbf{W}_i, \mathbf{C}_i = \mathbf{H}_i \mathbf{B}_i, \gamma = \beta \alpha$, 于是 \mathbf{C}_i 取值属于 $\{+1, -1\}$, 重写式 (26.35) 如下：

$$\gamma^*, \mathbf{C}^* = \arg \min_{\gamma, \mathbf{C}} \|\mathbf{1}^T \mathbf{Y} - \gamma \mathbf{1}^T \mathbf{C}\| \quad (26.36)$$

所以有：

$$\mathbf{C}^* = \text{sign}(\mathbf{Y}) = \text{sign}(\mathbf{X}^T) \text{sign}(\mathbf{W}) = \mathbf{H}^{*T} \mathbf{B}^* \quad (26.37)$$

$$\gamma^* = \frac{\sum |Y_i|}{n} = \frac{\sum |X_i| |W_i|}{n} \approx \left(\frac{1}{n} \|\mathbf{X}\|_{l_1} \right) \left(\frac{1}{n} \|\mathbf{W}\|_{l_1} \right) = \beta^* \alpha^* \quad (26.38)$$

二值网络的收益体现在存储和计算两个方面。与双精度的网络相比, 二值网络的存储是其 1/64。在 CPU 计算方面, 二值参数网络采用加减代替了乘加, 二值参数二值输入网络采用位运算代替了大量乘加, 计算量进一步下降。参考文献 [20] 中指出, XNOR-Net 在卷积层

的运算速度能够达到 58 倍。

当训练集较小时,二值网络能够取得跟实数网络近似的准确率。然而,当训练集增大时,二值网络的表征能力就不如实数网络了。参考文献 [20] 中指出,在 ImageNet 上,二值参数网络的 Top-1 准确率比实数网络低 2.8%,XNOR-Net 的 Top-1 准确率比实数网络低 14.4%。

26.7 小结

前面几节介绍了几种典型的模型压缩方法,但模型压缩的方法还有很多,在实际应用中,人们也常使用量化压缩的方法,即将连续的参数值近似为整数值,或者将大量可能的离散取值近似为较少的离散值。从本质上看,模型压缩可以视为一种特殊的正则,运用得当时,不仅能够降低模型的复杂度,还能够提升模型的精度。而且通过对模型压缩的研究,也能给我们带来一些提升模型精度的方法。比如,当分析各层网络对剪枝的敏感度时,可以通过增加比较敏感的层的参数数量来获得较高的模型精度。再比如,在不降低模型精度的基础上通过剪枝压缩网络后,再将其扩充成稠密网络,往往能够提高模型的精度。

此外,随着移动端模型越来越受关注,这方面的最新研究也非常值得追踪,比如 Distilling^[21]、MobileNet^[22] 和 ShuffleNet^[23]。关于这些模型的细节,本文不再详细展开介绍,感兴趣的读者请自行参考相关文献。

参考文献

- [1] A. Krizhevsky, I. Sutskever, G.E. Hinton. Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. (2012) 1097-1105.
- [2] S. Han, H. Mao, W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding[J]. CoRR, abs/1510.00149, 2015, 2.
- [3] Y.N. Dauphin, Y. Bengio. Big neural networks waste capacity. arXiv preprint arXiv:1301.3583 (2013).
- [4] J. Ba, R. Caruana. Do Deep Nets Really Need to be Deep?[J]. Advances in Neural Information Processing Systems, 2014:2654-2662.
- [5] B. Poole, S. Lahiri, M. Raghu, et al. Exponential expressivity in deep neural networks through transient chaos[J]. arXiv preprint arXiv:1606.05340, 2016.

- [6] Y. LeCun, J.S. Denker, S.A. Solla, et al. Optimal brain damage. *Advances in Neural Information Processing Systems(NIPS)*. 1989, 2: 598-605.
- [7] C.A. Walsh. Peter Huttenlocher (1931-2013). *Nature*, 2013, 502(7470): 172-172.
- [8] B. Hassibi, D.G. Stork. Second order derivatives for network pruning: Optimal brain surgeon[M]. Morgan Kaufmann, 1993.
- [9] J. Yosinski, J. Clune, Y. Bengio, et al. How transferable are features in deep neural networks? *Advances in neural information processing systems(NIPS)*. 2014: 3320-3328.
- [10] S. Tang, J. Han. A pruning based method to learn both weights and connections for LSTM[J].
- [11] S. Han, J. Pool, J. Tran et al. Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems(NIPS)*. 2015: 1135-1143.
- [12] W. Chen, J.T. Wilson, S. Tyree, K.Q. Weinberger, Y. Chen. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788* (2015).
- [13] M. Lin, Q. Chen, S. Yan. Network in network[J]. *arXiv preprint arXiv:1312.4400*, 2013.
- [14] C. Szegedy, W. Liu, Y. Jia, et al. Going deeper with convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*. 2015: 1-9.
- [15] K. He, X. Zhang, S. Ren, et al. Deep residual learning for image recognition[J]. *arXiv preprint arXiv:1512.03385*, 2015.
- [16] F.N. Iandola, M.W. Moskewicz, K. Ashraf, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 1MB model size[J]. *arXiv preprint arXiv:1602.07360*, 2016.
- [17] K. He, J. Sun. Convolutional neural networks at constrained time cost. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*. 2015: 5353-5360.
- [18] M. Courbariaux, Y. Bengio, J.P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in Neural Information Processing Systems*. 2015: 3123-3131.
- [19] M. Courbariaux, Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1[J]. *arXiv preprint arXiv:1602.02830*, 2016.
- [20] M. Rastegari, V. Ordonez, J. Redmon, et al. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks[J]. *arXiv preprint arXiv:1603.05279*, 2016.
- [21] Hinton G, Vinyals O, Dean J. Distilling the knowledge in a neural network[J]. *arXiv preprint arXiv:1503.02531*, 2015.

[22] Howard A G, Zhu M, Chen B, et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications[J]. arXiv preprint arXiv:1704.04861, 2017.

[23] X Zhang, X Zhou, M Lin, J Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. arXiv preprint arXiv:1707.01083.

27

增强学习

AlphaGo 将人工智能推到风口浪尖上，而 AlphaGo 用到的最主要技术就是增强学习和深度学习。本章首先具体介绍增强学习的基本原理及相关技术，然后探索 AlphaGo 的具体算法。

27.1 什么是增强学习

学习是每个人都会遇到的问题，也是伴随人一生的事情。现在的人可以选择从书本、学校甚至网络课堂学到各种各样的知识，但是对于原始人来说，他们学习的主要方式就是和自然界进行交互。他们是如何与自然界进行交互的呢？很久很久以前，当我们的祖先行走在森林中时，他们会自然而然地产生一些行为，如挥动自己的手臂、捡起地上的东西等。当他们完成这些动作之后，总能捕捉到一些周围环境带给他们的信息与反馈，将这些信息和刚才发生的动作联系在一起，就会使他们明白这些动作和周围环境的关系，以及下一步甚至未来应该做些什么，等等。比如挥动手臂后被某个野兽发现，野兽会发动攻击，对人类的安全造成威胁；摘下树上的水果可以得到补充能量的食物。有些行为会对人产生正面的影响，有些行为会对人产生负面的影响。这些都可以算作大自然对人类行为的反馈。正是通过这种不断地和大自然交互的学习方法，祖先们逐渐掌握了许多生存技能。

不仅在原始社会，对于现在的社会，我们每天同样会进行很多这样的交互。当我们操纵电脑时，每发出一个行为，电脑都会做出一定的响应，比如通过点击鼠标打开了某个文件、

播放了某个音乐等。通过这些响应，我们就逐渐明白每一个动作所产生的效果，也逐渐明白在各种场景下应该采取的动作。这种学习方式也是交互式的。

这种交互式的学习方法就是增强学习（Reinforcement Learning，也翻译成强化学习），可以说这种学习方法在我们的生活中无处不在。上面用举例的方式介绍了增强学习的概念，下面将对增强学习的过程进行详细的定义。

首先我们要明确增强学习中的两个主要角色——学习者和环境。我们将参与学习的本体称为学习者（Agent），比如上面例子中的原始人；而与 Agent 进行交互的外部环境称为环境（Environment），在上面的例子中，大自然以及其他动物可以统称为环境。为了方便描述和计算，我们将流动的连续时间抽象成一个个离散的时间片段，将 Agent 和环境双方的交互限制在一个个时间片段内；每一个时间片段对应双方的一轮交互。具体来说，双方会产生如下交互。

（1）在时间窗口 t 内，环境会处于某一个状态（State） S_t 下，而 Agent 也可以感受到这个状态。比如原始人处于一片森林中，在他的面前有一只死兔子。

（2）Agent 会根据自己的某种应对环境的策略（Policy） π 对环境所处的状态产生某个动作（Action） A_t 。比如原始人的策略是看到死动物就上前捡走。

（3）环境会根据 Agent 在上一个时间窗口的动作 A_t 产生一定的反馈，以奖励（Reward） R_t 的形式被 Agent 捕获或者感知。与奖励一道被 Agent 捕获或感知的，还有环境的最新状态 S_{t+1} 。当然，这里的“奖励”并不是中文中带有褒义的奖励，这个“奖励”有可能是正面的奖励，也有可能是负面的奖励。比如原始人捡起死兔子，获得了食物，那么这个“奖励”就是正面的奖励；如果原始人捡起兔子，却掉入了一个陷阱中，被陷阱中的锐器刺死，那么这个“奖励”实际上就是一个惩罚，也就是负面的奖励。

就这样，在每一个我们划分的时间片段内，上面的过程将会进行一次，整个过程也会随着时间的流逝不断循环进行，直到某个终止条件出现，比如上面所说的原始人死了，那么他和环境的交互就终止了。整个过程可以用图 27-1 来表示。

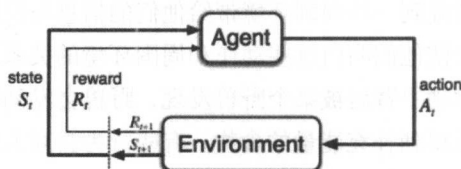


图 27-1 增强学习交互图^[1]

Agent 作为交互过程中的主动方，他的目标是尽可能多地获得回报。关于对这个问题的详细讨论，我们将在后面进行介绍。上面关于原始人的例子是一个比较简单而且粗糙的例

子，在现代社会这样的交互式例子也非常多。比如电子游戏就是增强学习中一个很重要的应用场景，这里我们给出一个经典的游戏作为例子：俄罗斯方块，如图 27-2 所示。

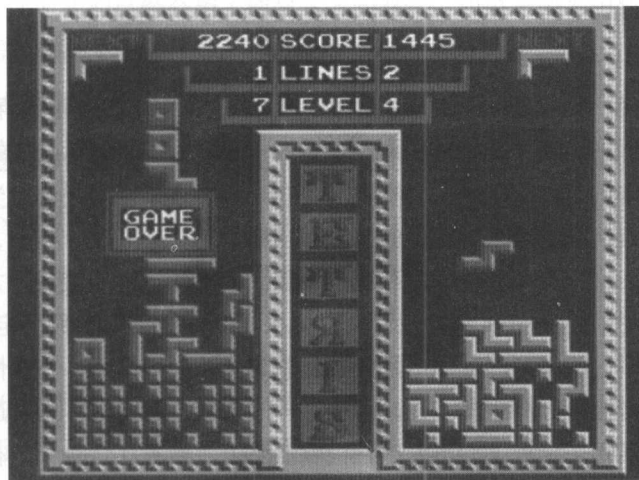


图 27-2 俄罗斯方块游戏（出处：<http://blog.csdn.net/bruesz/article/details/2200623>）

俄罗斯方块的游戏规则比较简单。在一个 $M \times N$ 的空间中，一些固定大小的方块组合会从空间的上方出现，以一定的速度向下降落。在降落的过程中，玩家可以旋转这些方块，也可以移动这些方块的水平位置。最终当方块降落到底部或者某个方块上面时，方块的位置将被固定。当空间内某一层的空间被方块装满时就会被消除并获得分数，同时装满越多层就会获得越多的分数。当方块充满整个空间且无法有新的方块进入空间时游戏宣告结束。游戏的目标是在结束前尽可能多地获得分数。游戏中玩家对每个方块的操作都会改变游戏当前的状态，同时会根据当前的状态（是否有消除的行）决定给予玩家多少分数。这个交互的过程和上面提到的增强学习的过程是一致的。

从上面的定义中可以看出，增强学习的模式本身有两个显著的特点：

1. 在不断的尝试中学习

Agent 并不会被告知自己要学习什么，而是要通过不断的实践获得经验。当 Agent 完成一个动作后，他将得到一些反馈，在增强学习中我们会将其抽象成一个数字，叫作奖励。对于 Agent 来说，他的目标就是尽可能地获得奖励，而为了实现这个目标，他需要做的是不断地尝试，获得反馈，然后利用反馈结果进行学习。对于俄罗斯方块这个游戏，在不断地游戏过程中，玩家会找到一定的规律，知道在给定的某种情况下应该做出什么样的操作。这个过程和我们熟知的监督学习并不相同，在监督学习中，Agent 要学习的内容和答案往往是事先

确定的，而且答案也是唯一的。如果把这种学习方法放在玩俄罗斯方块上，那么需要在操作俄罗斯方块每一步前都已经知道最优的操作，这显然是不现实的。

2. 延迟奖励

对于 Agent 来说，学习是一个时序的过程，Agent 要把这个时序问题看成一个整体，站在全局的角度看待问题，而不应该将问题拆解成一个个小的部分。Agent 的目标是尽可能在整个过程中获得最多的奖励，所以通常不但需要在当前状态下获得足够的奖励，还需要在未来的长期时间内获得奖励。未来将获得的奖励在当前状态下是无法获得的，所以相当于这部分奖励将延迟获得。这一点和监督学习相比也有很大的不同。对于俄罗斯方块这个游戏，最有说服力的例子就是它的得分机制。俄罗斯方块根据同时消去的层数决定应该得到的分数，同时消去的层数越多，所得到的分数的涨幅比例就越大，一般来说，一次性消除 4 行的得分会比 4 次各消除一行的得分高很多。正是因为这样的机制，使得很多玩家选择不急于消除方块，而是等方块积累到一定程度后再一次性消除。除俄罗斯方块之外，还有很多增强学习的例子具有这样的特点，延迟获得的奖励往往更诱人。

正是因为上面所述的两个特点，增强学习拥有其他学习方式所没有的一个挑战，那就是如何平衡探索（Exploration）和利用（Exploitation）两个策略的关系。

所谓探索，就是尽可能多地尝试新的动作，以获得更多不同的反馈和奖励，从而使 Agent 对问题有了更多的认识，并可以帮助他找到更好的方法。当然，在探索尝试的过程中，Agent 也会尝试一些奖励较低的方法，这样会拉低他获得的总奖励；在俄罗斯方块中，探索指的是采用一些全新的方块组合方式进行游戏，这些组合方式有的可能效果一般，有的则会获得比当前组合策略更多的分数。

所谓利用，就是根据现有的动作尝试和奖励信息，总结出一套在当前状态下最好的策略，从而认真贯彻这套策略，按照这套策略进行行动。这样做的好处是可以保证现在的动作是当前状态下最好的，可以获得最多的奖励；但是与此同时，Agent 也会失去获取更多奖励的潜在机会，因为 Agent 有可能还没有接触到比当前策略更好的策略，他还没有体会到更好的策略能为他带来更多的奖励。在俄罗斯方块中，利用指的是采用现有方块组合中最好的策略进行游戏，不去尝试新的组合策略。

27.2 增强学习的数学表达形式

在了解了增强学习的大致框架之后，我们继续深入探讨增强学习的数学形态。一个增强学习系统可以抽象成马尔可夫决策过程（Markov Decision Process, MDP）的模式，其中还包

括策略函数 (Policy Function)、奖励 (Reward) 与回报 (Return)、价值函数 (Value Function) 这几个核心部分。下面将详细介绍这几个部分。但是在介绍这些概念之前, 首先要引出介绍这些概念的原因。前面我们已经介绍了增强学习的概念, 如果希望采用数学的方式来描述这个交互过程, 则需要解决下面的问题。

- Agent 的动作是根据什么做出的?
- 这个部分是否可以通过模型表达出来? 环境是如何随 Agent 的动作变化的?
- 这里面是不是包含着某种规律? 如果有规律, 那么能不能用模型把它表达出来?

27.2.1 MDP

对于大多数的增强学习问题, 我们都可以用马尔可夫决策过程 (MDP) 来描述。MDP 这个概念可以分两个方面进行介绍。这里假设 Agent 在每一个状态下的动作是有限的, 同时环境状态的数量也是有限的。首先是交互过程的马尔可夫特性。我们知道增强学习的过程是 Agent 和 Environment 交互的过程, 当 Agent 的 Action 产生后, Environment 会切换到下一个状态, 那么这个新状态的产生会和哪些因素有关呢? 从直觉上看, 新状态 S_{t+1} 会与 S_t 和 A_t 相关。那么除此之外, 会不会和 S_{t-1} 和 A_{t-1} 相关呢? 拥有马尔可夫特性的问题告诉我们, 不会。马尔可夫特性将状态转移限定在只和前一个状态相关, 也就是说:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}$$

相当于:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\}$$

这个特性极大地简化了问题的复杂度, 使得整个问题变得更加清晰。同时, 状态间相互依赖的特性依然保留, 使得这个特性仍然可以保留问题本身最核心的一些特点。

其次是决策过程。当 Agent 接收到某一时刻 t 的 S_t 和 R_t 之后, 它将根据自身的策略做出相应的 A_t , Environment 接收到 Action 后, 会根据 S_t 和 A_t 转换到下一个状态 S_{t+1} , 转换的方式依赖于 $p(s_{t+1} | s_t, a_t)$ 概率, 同时给出相对应的奖励 $R_{t+1} = r(s_t, a_t, s_{t+1})$ 。之后 Agent 再做出选择。如此不断进行的状态转换构成了环境状态和 Agent 动作的序列, 这个序列就可以描述 MDP 过程。

27.2.2 策略函数

策略描述了 Agent 的行为方式。具体来说，就是描述了针对 Environment 的某个状态，Agent 将采取什么样的动作，而这个动作也是我们心目中最好的。所谓最好的动作，就是能让 Agent 获得最多奖励的动作。对于一些比较简单的问题，策略是一个简单函数或者一个“状态-动作”的查找表，每一种特定的状态都可以从查找表中找出对应的最优动作。而对于一些复杂的问题，采用查找表的方式就比较困难了。比如环境的状态空间非常大，对应 Agent 的动作也非常多，查找表很难存储所有的最优方案，这个时候策略函数也需要有更加复杂的表示方式，例如建立一个从环境到动作的函数。同时，对于 Environment 的某个状态，Agent 既可以给出确定的动作，也可以给出随机的动作。前面提到了增强学习中的一个挑战——探索与利用。在一些场景下，我们对当前“计算”得到的策略函数并没有百分之百的信心，也许对于某个状态，存在一个比策略函数更好的动作。在实际中，一些策略函数会以一个比较小的概率采取“探索”的策略，尝试最优策略之外的策略。这种方式往往能获得更好的效果。对于俄罗斯方块这个游戏，每个人的策略可能都有所不同，有的人会采用“尽可能得更多分”的策略，在“竖条”出现前，尽可能不去清除方块，而是积攒起来，等到“竖条”出现时再一并清除，以获得最多的分数；也有人为了稳妥起见，尽可能早地消除方块，保证自己立于不败之地，这都是策略的体现。

27.2.3 奖励与回报

奖励函数（Reward Function）描述了在某种环境状态下执行了某个动作之后能够获得的奖励的数目，数目的多少决定了当前行为在当前状态下的价值。这个函数一般可以被 Agent 直接使用。奖励函数一般具有时间不变性，也就是说，对于同一个状态或者同一个状态-动作对，其奖励不会因为它出现时间的不同而不同，其确定性也使得它可以更好地帮助 Agent 调整合适的策略。在增强学习中，奖励一般以数值的形式表示。这样的表示方式也非常常见，比如在大量的电子游戏中都有得分这样的机制，玩家采取一定的动作之后，游戏系统都会根据玩家的行为改变其分数，有的是增加有的是减少。这些得分很好地反映了玩家当前的表现，和奖励的定义非常相近。前面提到 Agent 的目标是尽可能多地获得奖励，这里将“尽可能多地获得奖励”换成“最大化长期奖励”，也就是最大化从当前状态到某种结束状态能获得的所有奖励。由于 Agent 的目标是长期获得最多的奖励，这和在当前状态下获得最大化奖励的目标不太一样。对于长期奖励这个目标，这里用回报（Return）来区别。那么 Agent 的目标就是最大化回报的数量。

根据“长期”的期限不同,增强学习可以分成两类:有明确结束状态的和没有明确结束状态的。

有明确结束状态的问题是比较常见的,比如电子游戏,一般都会有明确的结束状态(游戏通关或者主角死亡),那么对于这样的问题,Agent 和 Environment 的交互流程会有一个终点。

对于没有明确结束状态的问题,对于 Agent 和 Environment 的交互不会看到一个明显的终点,两者的交互会无限延伸下去。下面以守林人问题为例进行说明。在世界的某个角落,有一片森林,这片森林一直以来有一批守林人,他们的主要任务就是守卫这片森林。假设这片森林中种着的全是 N 年生的树木,也就是说,这批树木在生长 N 年后会完全长成。每一年的开始,这些守林人都要做出一个决策:是继续让这批树木生长,还是砍掉所有树木卖钱?假设这片森林的土地永远存在,而这批守林人经过更新换代,也一直存在,那么这就是一个在理想状态下没有明确结束状态的问题(关于这个问题,在后面的介绍中我们还会对其进行详细分析)。

对于上面提到的两类问题,回报的表达形式也不太相同。对于有明确结束状态的问题,回报相当于从当前状态到结束状态的奖励的加和,可以写作:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

对于无明确结束状态的问题,回报相当于从当前状态到无穷尽的未来所有奖励的和,可以写作:

$$G_t = \sum_i^{\infty} R_i$$

相对而言,对于有明确结束状态的问题,回报的数字是可以计算的(当然可能也会比较大);而对于无结束状态的问题,由于交互可能是无限的,如果按照上面的公式进行计算,这个数字可能是无穷大的。如果所有的回报值都是无穷大的,那么它们将无法进行比较,显然我们需要解决回报值无穷大的问题。解决这个问题的办法是引入一个变量:折现率。折现率的概念其实对我们来说并不陌生,可以用储蓄问题来解释这个概念。假设我们现在有 100 元钱,把这 100 元钱存到某个银行里(或者买了某家金融服务机构的理财产品),银行给出的年化利率是 5%,那么一年后这 100 元钱将变成 105 元。也就是说,在利率一定的情况下,一年后的 105 元等于现在手里的 100 元。所以如果要把未来的钱换算到当前时间下,则钱的数量需要减少。同理,如果站在当前时间段去计算未来的奖励,则未来的奖励同样需要折现到现在的时间下。我们用 γ 表示折现率,这个变量是小于 1 的,这样和上面描述的概念是符合

的。此时，对于回报的计算公式可以变成：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

可以看出，现在无明确结束状态问题的回报也变得有界了。我们可以做如下证明。假设每一个动作反馈的奖励是有界的，那么必然存在一个上界 R_{\max} ，于是有：

$$\begin{aligned} G_t &\leq R_{\max} + \gamma R_{\max} + \gamma^2 R_{\max} + \cdots \\ &= R_{\max} \sum_{k=0}^{\infty} \gamma^k \\ &= \frac{R_{\max}}{1 - \gamma} < \infty \end{aligned}$$

那 γ 的数值该如何设置呢？这和实际问题相关。对于一些需要有长远眼光的问题来说， γ 应该靠近 1；而对于一些不需要长远眼光的问题来说， γ 应该靠近 0；对于一些极限状况， γ 等于 0。那么这个问题就完全变成了“走一步，看一步”的问题，回报和奖励也就相等了。

27.2.4 价值函数

前面提到了奖励和回报两个概念，对于奖励，Agent 可以通过和环境进行交互来获得；而回报则无法直接获得。价值函数（Value Function）的作用就是计算回报这个数值，将当前环境所处的状态、状态-动作对与回报关联起来。一般来说，价值函数会以两种形式展现出来。

（1）状态价值函数（State-Value Function） $v_{\pi}(s)$ ，表示当环境处于 s 状态时的期望回报：

$$v_{\pi}(s) = E_{\pi}[G_t | S_t = s] = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$$

（2）动作价值函数（Action-Value Function） $q_{\pi}(s, a)$ ，表示当环境处于 s 状态且 Agent 采用了动作 a 后的期望回报：

$$q_{\pi}(s, a) = E_{\pi}[G_t | S_t = s, A_t = a] = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

27.2.5 贝尔曼方程

上面的状态价值函数是通过从当前状态开始无限延伸下去的奖励加和得到的，而实际上我们是无法得到未来所有状态的，但是可以对其进行变换使价值函数变得可解：

$$\begin{aligned}
 v_{\pi} &= E_{\pi}[G_t | S_t = s] \\
 &= E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right] \\
 &= E_{\pi}[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s] \\
 &= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_{t+1} = s'\right]] \\
 &= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v_{\pi}(s')]
 \end{aligned}$$

由此可见，对于当状态为 s 时，价值函数的取值可以通过其他状态的值求出，这个公式也被称为贝尔曼方程（Bellman Equation），是增强学习中十分重要的一个公式。同时，两个价值函数之间也可以建立起联系：

$$\begin{aligned}
 q_{\pi}(s, a) &= \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v_{\pi}(s')] \\
 v_{\pi} &= \sum_a \pi(a|s) q_{\pi}(s, a)
 \end{aligned}$$

27.2.6 最优策略性质

到目前为止，所有和增强学习相关的基本概念已经全部介绍完毕，相信读者对如何利用数学的方式来描述增强学习问题已经有了一定的认识。下面将介绍增强学习中一个十分重要的性质——最优策略性质。

在 Agent 与环境的交互过程中，Agent 当然希望自己能够获得最多的回报，那么就需要自己拥有最优的动作策略。我们可以给出最优策略的定义：对于一个策略函数 π ，如果它和其他的 π' 相比，对于任意一个 s ，都有 $v_{\pi}(s) \geq v_{\pi'}(s)$ ，那么就称这个策略 π 是最优的。对于最优策略的价值函数，我们定义它的状态价值函数为 $v^*(s)$ ，状态-动作价值函数为 $q^*(s, a)$ 。

与前面的推导过程类似，我们可以用贝尔曼方程对其进行推导：

$$\begin{aligned}
 v_*(s) &= \max_{a \in A(s)} q_{\pi^*}(s, a) \\
 &= \max_a E_{\pi^*}[G_t | S_t = s, A_t = a] \\
 &= \max_a E_{\pi^*}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \\
 &= \max_a E_{\pi^*}[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a] \\
 &= \max_a E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
 &= \max_{a \in A(s)} \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')]
 \end{aligned}$$

从上面的公式可以看出，当知道了最优价值函数后，就可以推导出最优的策略来。由于 $v^*(s)$ 是由最优的动作得到的，那么在公式中只要计算出所有 A_t 对应的状态价值，找出其中价值最大的那个，就是当前状态最优的策略。想要求出完整的策略，就需要把所有的状态遍历一遍。当然，对于那些状态很多的问题，求解最优价值函数已经十分困难，更别说遍历所有状态求解最优策略这种计算量很大的问题了。

27.3 用动态规划法求解增强学习问题

在介绍完增强学习的基本框架和其中的几个关键问题后，我们开始求解增强学习问题。本节主要介绍一些经典的求解算法，这些算法可以分成两类：已知环境模型和未知环境模型。

对于已知环境模型的问题，我们将介绍策略迭代法（Policy Iteration）和价值迭代法（Value Iteration），它们被统称为动态规划法。

对于未知环境模型的问题，我们将介绍蒙特卡罗法（Monte Carlo）和时序差分法（Temporal Differential）。

27.3.1 Agent 的目标

由于动态规划法需要知道环境模型的具体信息，因此在这里我们要对模型做出定义。Agent 在发出动作前已经知道了如下信息：

- 模型的状态转移概率 $p(S_{t+1} = s' | S_t = s, A_t = a)$ 。
- 模型的奖励函数 $r(s, a, s') = E[R_{t+1} | A_t = a, S_t = s, S_{t+1} = s']$ 。

根据上面的信息求解出最优策略 $\pi(A_t = a | S_t)$ 。当然，在前面的章节中我们曾经论证过，当获得了最优的价值函数后，就可以找到最优策略，这个过程可能会比较耗时间，但是一定可以实现。我们将利用这个性质进行求解。首先介绍第一种算法：策略迭代法（Policy Iteration），整个算法过程可以拆解成两个部分：策略评估（Policy Evaluation）和策略改进（Policy Improvement）。

27.3.2 策略评估

策略评估是指从策略到价值函数的推导。也就是说，对于某个策略 π ，求出利用这个策略所产生的价值函数 $v_\pi(s)$ 。利用贝尔曼方程，可以推导出求解价值函数的公式：

$$\begin{aligned}
 v_\pi(s) &= E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\
 &= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
 &= \sum_a \pi(a|s) \sum_{s'} p(s' | s, a) [r(s, a, s') + \gamma v_\pi(s')]
 \end{aligned}$$

由于我们假设已经知道了环境模型的全部信息，所以在上面的公式中唯一未知的就是价值函数。求解这个公式有两种方法：一种是求公式的闭式解，也就是把它当成一个方程组去解。如果模型中有 $|S|$ 个状态，那么这个方程组中就有 $|S|$ 个未知数。另一种是采用迭代的方式求解，由于上式中的 $\gamma < 1$ ，所以价值函数最终会收敛。因此可以将价值函数初始化成 0，然后反复利用上面的公式进行迭代，直到价值函数收敛为止。关于判断价值函数收敛的方法，我们可以比较前后两轮价值函数数值的差距，当差距小于某一个确定的值时，就认为价值函数已经收敛。有关迭代式的价值评估可以用下面的算法来进行计算。

算法 27-1 迭代式价值评估^[1]

输入：需要评估的策略 π

初始化 $v(s) = 0, \forall s \in S^+$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

temp $\leftarrow v(s)$

```


$$v(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) [r(s, a, s') + \gamma v(s')]$$


$$\Delta \leftarrow \max(\Delta, |\text{temp} - v(s)|)$$

until  $\Delta < \theta$  (一个很小的正数)
输出:  $v \simeq v_\pi$ 

```

27.3.3 策略改进

策略评估完成了从策略到价值函数的转化,这就为下一步——寻找更好的策略打好了基础。寻找更好的策略需要调整策略的内容,那么问题来了:当策略被调整后,价值函数会如何变化,是变大还是变小了呢?对于某个状态 s ,利用之前的策略,Agent 会采用动作 a ,最终状态 s 的价值函数是 $v_\pi(s)$ 。如果把动作 a 换成了 a' ,并在后面的动作中继续使用之前的策略,价值函数会发生怎样的变化呢?

$$\begin{aligned} q_\pi(s, a) &= E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s'} p(s' | s, a) [r(s, a, s') + \gamma v_\pi(s')] \end{aligned}$$

如果 $q_\pi(s, a')$ 比 $v_\pi(s)$ 大,那么证明对于状态 s ,采用 a' 这个动作会更好,反之则更差。如果 a' 更好,则可以推理出,对于所有出现状态 s 的场景,都可以用 a' 替换掉原来的策略。于是我们就可以给出策略改进理论的定义:令 π 和 π' 是一对确定的策略,如果对所有的状态 s ,都有:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

那么策略 π' 一定不弱于 π 。也就是说,采用 π' 评估的价值函数一定不会低于采用 π 评估的价值函数:

$$v_{\pi'}(s) \geq v_\pi(s)$$

这就是策略改进的方法,先对既有策略进行评估,然后尝试寻找每一个状态 s 可能的改进策略。

$$\begin{aligned} \pi'(s) &= \arg \max_a q_\pi(s, a) \\ &= \arg \max_a E[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg \max_a \sum_{s'} p(s' | s, a) [r(s, a, s') + \gamma v_\pi(s')] \end{aligned}$$

从公式中可以看出, 想让整体策略最优, 那么每一步的策略都必须和其他策略配合, 尽可能地保证最优。这和动态规划法中最优子结构的思想十分一致。所以策略改进的目标就是利用上面的公式找到更好的子部分最优策略, 然后利用动态规划法中的最优子结构的特性将它们拼接起来, 这样就完成了策略改进的工作。

27.3.4 策略迭代

当了解了上面提到的两个子步骤——策略评估和策略改进之后, 我们将这两个部分串联起来, 就得到了策略迭代的算法, 如下面的公式所示:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

其中的 E 步就是策略评估, 求出当前策略的价值函数; I 步就是策略改进, 根据当前的价值函数求出更好的策略。当策略不再改变时, 算法收敛结束。整体上, 策略迭代的算法可以由如下步骤表示。

(1) 初始化

$$v(s) \in R, \pi(s) \in A(s), \forall s \in S$$

(2) 策略评估

Repeat

$$\Delta \leftarrow 0$$

For each $s \in S$:

$$\text{temp} \leftarrow v(s)$$

$$v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s)) [r(s, \pi(s), s') + \gamma v(s')]$$

$$\Delta \leftarrow \max(\Delta, |\text{temp} - v(s)|)$$

until $\Delta < \theta$ (一个很小的正数)

(3) 策略改进

$\text{policy_stable} \leftarrow \text{true}$

For each $s \in S$:

$$a \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \arg \max_a \sum_{s'} p(s'|s, a) [r(s, a, s') + \gamma v(s')]$$

If $a \neq \pi(s)$, **then** $\text{policy_stable} \leftarrow \text{false}$

If policy_stable, then stop and return v and π ; else go to 2

27.3.5 策略迭代的例子

下面我们将用一个具体的例子来展示策略迭代算法的效果。这里将使用前面提到的守林人问题作为例子。首先对守林人问题进行具体化，以方便计算。

(1) 森林中的树木是 4 年生的，即树木前 4 年稳定成长，4 年后不再成长。

(2) 当采取的动作作为保护时，森林的状态转移概率如下：

[[0.1 0.9 0. 0.]

[0.1 0. 0.9 0.]

[0.1 0. 0. 0.9]

[0.1 0. 0. 0.9]]

其中行代表当前森林状态，列代表转移到的森林状态，状态是指对应的森林树木成长了几年，比如第 0 行表示新种植的，第 1 行表示成长了 1 年，第 2 行表示成长了 2 年，依此类推。第 2 行第 3 列的数字 0.9 表示当动作为保护时，2 年的森林有 0.9 的概率转移（长成）到 3 年的森林。当然，有 0.1 的概率会转移到 1 年的森林，因为森林可能遭遇大火。

(3) 当采取的动作作为砍伐时，森林的状态转移概率如下：

[[1. 0. 0. 0.]

[1. 0. 0. 0.]

[1. 0. 0. 0.]

[1. 0. 0. 0.]]

和上面的状态转移概率类似，由于采取了砍伐的策略，所以森林一定会回到起点的。

(4) 对于保护和砍伐，对应的奖励值如下：

[[0. 0.]

[0. 1.]

[0. 1.]

[1. 10.]]

其中行表示森林状态，第 1 列表示保护带来的奖励，第 2 列表示砍伐带来的奖励。从数字中可以看出，最大的奖励出现在砍伐已成长 4 年树木的时候，这给我们的直觉符合，其他的动作显然不会带来更多的收益。

通过上面的描述,环境模型已经完全被我们所知,那么接下来就可以直接利用状态转移概率和奖励值进行计算了。为了方便讲述算法流程,我们将状态转移概率矩阵定义为 $p(a, s_t, s_{t+1})$, 奖励矩阵定义为 $r(s_t, a)$ 。策略中的 0 表示保护森林, 1 表示砍伐森林。我们初始化状态-动作价值函数中所有的变量为 0。首先进行策略评估, 利用贝尔曼方程对价值函数 $q(s, a)$ 进行更新: $q(s, a) = \sum_{s'} p(s, a, s') * (\gamma * V(s') + r(s, a))$, 于是可以得到:

$$q(s, a) = \begin{bmatrix} 0.0 & 0.0 & 1.0 \\ 0.1 & 1.0 & 10.0 \end{bmatrix}$$

然后进行策略改进, 对于每一个 s , 求出最优的动作作为当前的策略:

$$a' = \arg \max_{a'} q(s, a')$$

$$\pi = [0 \ 1 \ 1 \ 1]$$

这样就完成了一轮迭代, 接下来使用所产生的策略继续进行策略评估和策略改进, 最终可以得到价值函数和策略:

$$q(s, a) = \begin{bmatrix} 17.72923206 & 19.91802614 & 22.62024106 & 23.62024106 \\ 15.95630885 & 16.95630885 & 16.95630885 & 25.95630885 \end{bmatrix}$$

$$\pi = [0 \ 0 \ 0 \ 1]$$

最终的策略告诉我们, 只有在森林到达第 4 年时去砍伐, 而其他年都应该保护, 才能最大化长期回报。通过策略评估和策略改进的交替计算, 我们就求出了最优的价值函数和策略函数。

27.3.6 价值迭代

策略迭代存在一个问题, 就是在做策略评估的过程中, 为了保证价值函数可以收敛, 需要做很多轮的迭代计算, 这实际上也比较消耗时间。那么一定要等待它完全收敛吗? 如果在不完全收敛的情况下就停止策略评估的过程会不会有问题呢? 实际上提前停止迭代是可以的, 不需要等到价值函数完全收敛, 只要保证它有一定的更新即可。价值迭代就是基于这样的思想产生的, 它的算法结构和策略迭代完全相同, 但是在策略评估过程中选择了最为激进的策略——只做一轮迭代就结束。算法如下:

算法 27-2 价值迭代算法

随机初始化数组 v (例如: $v(s) = 0, \forall s \in S^+$)

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

temp $\leftarrow v(s)$

$v(s) \leftarrow \max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$

$\Delta \leftarrow \max(\Delta, |\text{temp} - v(s)|)$

until $\Delta < \theta$ (一个很小的正数)

输出: 满足如下条件的策略 π

$$\pi(s) = \arg \max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$$

我们可以从策略迭代的角度来理解这个算法,也就是上面提到的思路。同样可以用动态规划法的思想对算法进行解释。首先将所有状态的价值设为 0,同时将增强学习的时间调整到学习的结束时间,对于有明确结束状态的问题,我们就把时间设为结束时间 T ;对于无明确结束状态的问题,我们假定一个“世界末日”时间,这一点的时间为 T ,如图 27-3 所示。

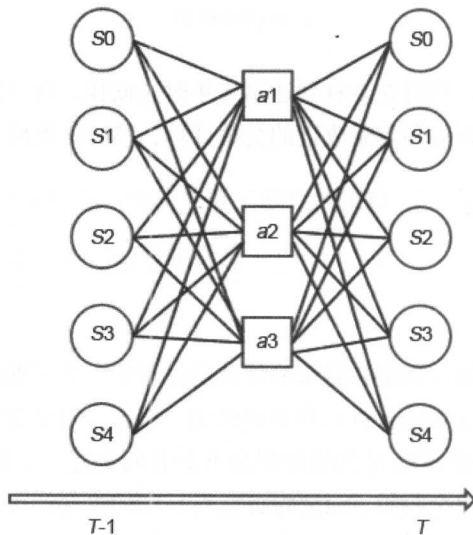


图 27-3 价值迭代法示意图^[1]

我们知道了在 T 时刻每一个状态的价值，那么 $T-1$ 时刻的最优价值是多少呢？根据公式可以计算出来：

$$v_{T-1}(s) = \max_q q(s_T, a) = \max_a \sum_{s'} p(s, a, s') * [\text{reward}(s_T, a) + \gamma * v_T(s')]$$

首先需要计算每一个 $q(s_T, a)$ 的值，然后对于每一个状态 s 取出其中最大的 $q(s_T, a)$ 得到 $v_{T-1}(s)$ ，这就是在 $T-1$ 时刻的最优价值。同时我们也可以求出在 $T-1$ 时刻的最优策略，因为已经知道这一时刻的最优价值函数，自然可以推导出最优策略。计算的过程如图 27-3 所示。同理，使用同样的方法还可以推算出 $T-2$ 时刻的最优状态，并不断推算下去。由于 $\gamma < 1$ ，不断推算下去，价值函数最终会趋向于收敛，因此当价值函数收敛时，也就是求得最优价值函数时，这时求出的策略就是最优策略。从上面的公式可以看出，公式中并没有与策略相关的内容，所以只需要在价值函数收敛后求出相应的策略即可。在这样的描述过程中，我们可以看到算法的主角变成了价值函数，与前面策略迭代里面的“策略评估-策略改进”不同，在价值迭代过程中，每一轮都是“当前最优价值计算-策略反推”，策略是根据价值得到的，所以这个算法也被称为价值迭代法，而不是“一轮评估式策略迭代法”。

27.3.7 价值迭代的例子

这里还是用守林人问题做例子。采用同样的参数，我们首先得到 $T-1$ 时刻的 $q(s, a)$ 的值：

[[0. 0. 0. 1.]

[0. 1. 1. 10.]]

求出其中最大的值，可以得到：

[0. 1. 1. 10.]

继续进行计算，可以得到 $T-2$ 时刻的 $q(s, a)$ 的值：

[[0.81 0.81 8.1 9.1]

[0. 1. 1. 10.]]

采用同样的方法进行计算，最终价值函数 $v(s)$ 的改动会小于期望值，此时函数收敛：

[[17.52988088 19.71846912 22.42054601 23.42054601]

[15.75632729 16.75632729 16.75632729 25.75632729]]

这时可以根据最优价值函数求出最优策略：

$(0, 0, 0, 1)$

可以看出，使用价值迭代法求出的策略与使用策略迭代法求出的策略完全一致，两者的价值函数也大体一致。

27.3.8 策略函数和价值函数的关系

从前面的两种算法中可以看出，策略函数和价值函数在求解中都起到关键性作用，那么两者的关系如何呢？从策略迭代法中可以看出，当策略函数固定时，数需要求解价值函数，当价值函数固定时，策略函数又可以变动；在价值迭代法中，当策略函数固定时，可以求出新的当前最优价值函数，当价值函数固定时，新的策略又可以被确定下来。所以两种算法本质上都是采用固定其中一个优化另一个的方式进行的，只是在方法上稍有不同，但在思想上是完全一致的。而这个思想就是所谓的“泛化策略迭代”（Generalized Policy Iteration），如图 27-4 所示。

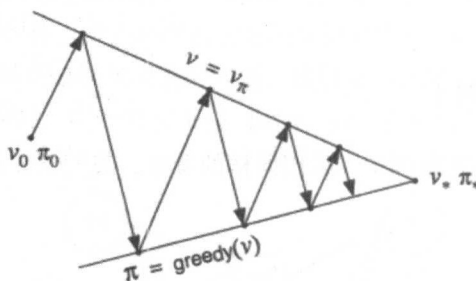


图 27-4 泛化策略迭代示意图^[1]

根据这样的思路，我们可以灵活地调配优化策略和优化价值这两部分，使得算法更好地完成收敛。比如可以采用两轮迭代的策略迭代法，这样策略评估的计算会更精确一些，由此带来的整体的迭代轮数也会相应地减少一些。一旦构建出这样一个泛化策略迭代的框架之后，我们就可以根据实际问题的需要调整价值函数和策略函数各自计算的权重了。

27.4 无模型算法

前面讲到的基于动态规划法的算法有一个特点，那就是它们都需要明确知道环境模型的具体信息，其中包括状态之间的转移概率和状态转移的奖励，相当于 Agent 已经把环境了解

得清清楚楚，所以给出最优的策略，实现最大化长期回报是较为可行的。而接下来要介绍的算法则没有这么好的条件。对于绝大多数现实中的增强学习问题，Agent 只能在采取动作之后获得下一个状态以及对应的奖励，对其中的概率转移函数和奖励函数的细节无法知晓。这一类算法也被称为无模型（Model-Free）算法。当然，没有模型不代表无法进行求解，我们可以通过采样的方法进行求解，也就是具有增强学习特色的学习方式——在实践中学习，当模型通过实践采集到大量的与环境交互的数据之后，它就可以利用这些数据求出价值函数的模型，求解在当前数据下的最优策略了。在无模型算法中，比较经典的算法有蒙特卡罗法和时序差分法。下面就来看看这两种算法。

27.4.1 蒙特卡罗法

在很多增强学习问题中，我们发现想获得环境模型完整的状态转移概率比较困难，但是获取每一步动作后的状态和奖励序列是比较容易的。比较典型的例子就是电子游戏，一般来说，玩家想要把每一个场景（也可以称作状态）下经过任意操作后产生的状态完全收集到是不太现实的，但是游戏会自然而然地为玩家展现出它的下一个场景和奖励。这里通常只考虑有明确终止状态的问题，这样将每一轮 Agent 从开始到结束的 MDP（马尔可夫决策过程）称为一个交互流（Episode）。通过和环境的不断交互，Agent 可以获得一定数量的交互流，并利用这些交互流学习策略。实际上蒙特卡罗法的总体计算思路和动态规划法的计算思路相似。对于蒙特卡罗法，首先要做的同样是策略评估，也就是对于固定的策略 π ，计算 v_π 和 q_π ，然后再进行策略改进。唯一不同的是，蒙特卡罗法在策略评估这一步采用了蒙特卡罗采样的方法。蒙特卡罗法的核心思想就是将交互流中顺序出现的状态和动作进行平均，从而得到对于每一个状态的价值函数，然后利用这个价值函数求出相应的策略。蒙特卡罗法在统计样本的过程中有两种方式：

- 全样本统计法（Every-visit MC method），也就是把每一个交互流中出现的所有状态都对记录下来，用于评估状态的价值函数。
- 首次出现样本统计法（First-visit MC method），也就是对于每一个状态 s ，在每一个交互流中只统计第一次出现的状态信息，用于评估状态的价值函数。

以下是蒙特卡罗版的策略评估算法，这里采用首次出现样本统计法。

算法 27-3 蒙特卡罗策略评估算法（采用 First-visit MC method）

初始化：

$\pi \leftarrow$ 需要评估的策略

$V \leftarrow$ 任意状态-值函数

$\text{Returns}(s) \leftarrow$ 空列表, 针对所有的 $s \in S$

Repeat forever:

(a) 使用策略 π 生成一个交互流

(b) 针对交互流中出现的每一个状态 s

$G \leftarrow s$ 第一次出现时的回报

将 G 添加在列表 $\text{Returns}(s)$ 的最后

Append G to $\text{Returns}(s)$

$V(s) \leftarrow \text{average}(\text{Returns}(s))$

上面的算法评估的是状态价值函数, 在拥有模型详细信息的情况下, 利用它推导出策略是没有问题的; 但是在没有模型信息的情况下, 计算策略需要从每一个状态出发, 遍历所有的动作计算出 $q(s, a)$, 然后再求出最优策略 $\arg \max q(s, a)$ 。既然这样, 我们需要考虑评估 $q(s, a)$ 的蒙特卡罗法。同样可以采用评估 $v(s)$ 的方法来评估 $q(s, a)$, 但是这里同样有一个问题, 那就是有的 (s, a) 对可能极难出现在交互流中。对于没有出现的 (s, a) 对来说, 计算它们的价值比较困难, 因此判断这个动作是否是这个 s 的最优策略变得比较困难。对于这个问题, 一种解决方法就是保持探索 (Exploration), 给所有没有在交互流中出现的动作一定的概率。在完成评估后, 我们就可以求出当前价值函数下的策略, 也就是策略改进的步骤。完整的算法如下:

算法 27-4 加入探索的蒙特卡罗策略评估算法

初始化:

for all $s \in S, a \in A(s)$:

$Q(s, a) \leftarrow$ 任意值

$\pi(s) \leftarrow$ 任意值

$\text{Returns}(s, a) \leftarrow$ 空列表

Repeat forever:

(a) 选择 $S_0 \in S$ 和 $A_0 \in A(S_0)$, 这样可以保证所有的 pair 概率都大于 0

(b) 根据策略 π 生成一个交互流

(c) 针对出现在交互流中的每一个 $\langle s, a \rangle$ 对

$G \leftarrow \langle s, a \rangle$ 第一次出现时对应的回报

将 G 添加在列表 $\text{Returns}(s)$ 的最后

$$Q(s, a) \leftarrow \text{average}(\text{Returns}(s, a))$$

(d) 针对交互流中的每一个状态 s

$$\pi(s) \leftarrow \arg \max_a Q(s, a)$$

27.4.2 时序差分法

时序差分法 (Temporal Difference) 结合了动态规划法和蒙特卡罗法的特点。和动态规划法类似, 时序差分法的参数更新只需要交互流中的部分信息即可, 不需要像蒙特卡罗法那样收集完一个交互流之后才能进行计算; 同时和蒙特卡罗法类似, 时序差分法同样根据交互流的信息进行参数更新, 而不像动态规划法那样需要知道模型的细节信息。在蒙特卡罗法中, 需要等待一个交互流结束后, 计算在这个交互流中某个状态 s 的回报 G_t , 然后用这个回报更新状态 s 的价值函数, 公式如下:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

式中的 α 指的是更新步长。与蒙特卡罗法不同的是, 时序差分法只需要等待下一个状态产生即可。利用当前价值函数对下一个状态的评估 $V(S_{t+1})$ 和动作产生的奖励 R_{t+1} , 就可以组合得到状态 S_t 的回报。于是可以得到如下公式:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

这就是时序差分法的更新公式。从另一个角度考虑这个公式, 可以将公式进行转换, 得到:

$$V(S_t) \leftarrow (1 - \alpha)V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1})]$$

实际上 α 相当于一个滑动平均的参数, 让最新的价值函数值由原有的价值函数值和计算得到的回报加权平均得到。

以下就是时序差分法的策略评估过程。

算法 27-5 时序差分法的策略评估

输入: 需要评估的策略 π

随机初始化 $V(s)$ (例如: $V(s) = 0, \forall s \in S^+$)

Repeat (for each episode):

初始化 S

Repeat (for each step of episode):

$A \leftarrow$ 策略 π 在 S 时采用的动作

采取动作 A , 得到奖励 R 及下一状态 S'

$$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$$

$S \leftarrow S'$

until S 是结束状态

时序差分法和蒙特卡罗法相比有一定的优势, 它可以自然地实现为在线增量的算法, 不必等到整个交互流结束后才更新价值函数, 而且在很多实际应用中, 时序差分法比蒙特卡罗法拥有更快的收敛速度。

27.4.3 Q-Learning

在增强学习中一个十分重要的算法是 Q-Learning, 它是一种 Off-Policy 的时序差分算法, 它的基本公式如下:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Q-Learning 的主要特点是每一次都把之前已更新的模型应用在新的评估中。该算法的具体过程如下:

算法 27-6 Q-Learning 算法

随机初始化 $Q(s, a), \forall s \in S, a \in A(s)$, 并初始化 $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

初始化 S

Repeat (for each step of episode):

使用由 Q (例如 ϵ -greedy) 衍生的策略, 针对状态 S 选择动作 A

采取动作 A , 得到奖励 R 及下一状态 S'

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$;

until S 是结束状态

27.5 Q-Learning 的例子

随着深度学习的不断发展及其在语音和图像上的成功应用,增强学习通过深度学习实现了重大突破。下面就来谈谈深度学习在增强学习上的应用。我们以 DeepMind 在 Atari 2600 游戏上的增强学习应用为例^[3]。

Atari 2600^[4] 是雅达利 (Atari) 在 1977 年 10 月发行的一款游戏机,当年风行一时,成为电子游戏第二世代的代表主机。其中经典的游戏包括 Adventure、碰碰弹子台、爆破彗星和 Pac-Man 等。游戏界面如图 27-5 所示。我们的目标是训练一个模型,并让模型代替人类去玩游戏,终极目标自然是让模型玩得比人类还好,同时具备玩游戏的通用技巧。

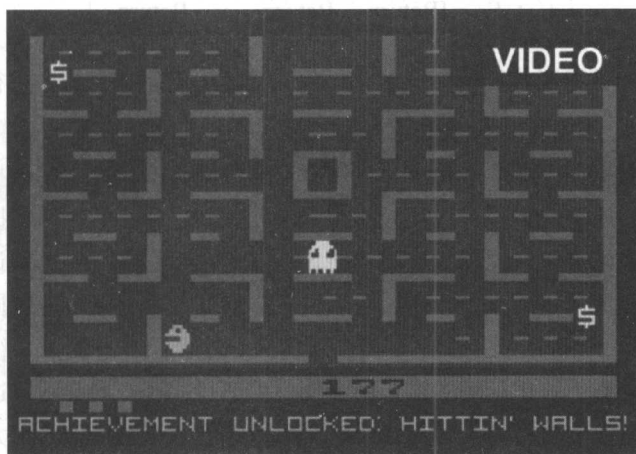


图 27-5 Atari 2600 中的 Pac-Man 游戏^[6]

首先,我们来分析要解决的问题是什么。这是一个无模型的问题,也就是说,我们并不知道这个游戏的完整状态是什么样子的,无法清楚地知道下一秒中怪物会在哪里出现。所以直接站在“上帝”的视角俯视这个游戏是不可能的,需要用无模型的方法进行学习。其次,深度学习的模型一般比较复杂,参数的数量比较多,而训练复杂的模型需要大量的训练数据,所以寻找大量有效的训练数据也是需要解决的问题。接下来,深度学习虽然可以解决提取特征的问题,但是如何对原始游戏信息进行处理也是一个有技巧的问题。最后,视频游戏的实时性很强,我们需要快速地对周围的环境做出反应,这是在工程上需要考虑的问题。下面就来解决上面的问题。

我们采用 Q-Learning 算法求解模型,也就是根据 $Q(s, a)$ 建模,求解公式如下:

$$Q^*(s, a) = E_{s' \sim \epsilon} [r + \gamma \max_{a'} Q^*(s', a') | s, a]$$

从公式的结构上看,模型的输入有两个部分: s 和 a , 输出是一个数字, 代表在当前状态下执行某个动作后的价值。如果直接采用这样的策略进行建模, 则会存在一些问题。首先, 模型输入两个部分的维度不同, 对于设计模型来说也有点难度; 其次, 将动作作为输入, 模型该如何表达也是一个问题。因此, 我们需要对模型进行一些转换。由于对于每一个动作 a 计算 $Q(s, a)$ 时, 环境状态 s 是不变的, 同时动作的数量也是固定的, 因此可以将模型的输入改为只有环境的输入, 而输出为所有动作的价值。所以要建立的模型就从原来的:

$$f : S \times A \rightarrow \text{Return}$$

变为:

$$f' : S \rightarrow [\text{Return}_{a1}, \text{Return}_{a2}, \dots, \text{Return}_{an}]$$

这样不论动作的数量形式发生任何变化, 我们都可以使用这个模型进行运算。模型拥有了很强的适应性。

其次就是训练数据的问题。理论上, 我们可以通过模拟产生大量的数据, 然后对这些数据进行训练。但是一局游戏的过程往往具有很强的相关性, 后面的一帧数据和前面的一帧数据十分相近, 因此如果直接对这样的数据进行训练, 则模型很有可能会被误导。比如把顺序排列的一组数据收集起来进行训练, 而这一组顺序排列的数据往往对应着同一个动作(比如不做任何动作), 那么训练的结果就是模型会学着采取这个操作; 而下一组顺序排列的数据会对应着另外一个动作, 那么训练的结果就会使模型向另一个方向学习。这样学习出来的模型显然是不好的。所以在进行增强学习训练时, 可以建立一个数据池, 把大量的数据预先存入这个数据池中, 然后再从这个数据池中随机抽取一定数量的数据集进行训练, 这样就可以保证训练数据不会错误地学习到一些动作模式。

接下来是特征抽取的问题。为了保证深度学习模型的普适性, 我们只使用图像的像素作为输入。同时为了让游戏中的一些动态信息得以保存, 我们将多帧图像合并起来作为输入, 这样输入将会包含游戏中的一些动作, 动态的问题也得到了解决。

最后是实时性的问题。正如上面所说的, 游戏存在一些相关性的问题, 所以不需要对每一帧都进行一个操作, 只需要每隔一段时间进行一个操作就可以了。

将上面的方案整理起来, 就得到了最终的算法方案。

算法 27-7 基于经验池的深度 Q-Learning 算法

将经验池 D 的大小初始化为 N

随机初始化动作价值函数 Q

for episode= 1 to M **do**

初始化序列的起始状态 $s_1 = x_1$ ，同时对其进行预处理，得到 $\phi_1 = \phi(s_1)$

for $t = 1$ **to** T **do**

在采取动作时，以 ϵ 的概率采取随机动作 a_t ，否则选择当前动作价值函数评估的最优动作 $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

在模拟器中执行动作 a_t ，并接收到返回的奖励 r_t 和新图像 x_{t+1}

设置新状态 $s_{t+1} = s_t, a_t, x_{t+1}$ ，同时进行预处理，得到 $\phi_{t+1} = \phi(s_{t+1})$

将数据 $(\phi_t, a_t, r_t, \phi_{t+1})$ 存储到经验池 D 中

// 训练

从 D 中随机采样一批数据： $(\phi_j, a_j, r_j, \phi_{j+1})$

利用 Q-Learning 的公式对采样出的数据计算期望的价值，对于终止状态， $y_j = r_j$ ；对于非终止状态， $y_j = r_j + \gamma \max_{a'} Q(\phi_{j+1}, a', \theta)$

采用梯度下降的方法，使用数据进行训练，更新动作价值函数模型

以上就是深度 Q-Learning 的一些细节问题。通过学习，模型可以很好地完成很多游戏，甚至在有些游戏上超越人类。除此之外，深度增强学习还表现出更为普适性的一面——对于不同种类的游戏，不需要建立新的模型，不需要针对不同的模型做不同的适配；同时，模型在训练时不需要游戏本身的知识和启发式方法的辅助，只需要将游戏画面输入，就可以较好地完成游戏，这使得对通用智能的探索又向前迈进了一步。

27.6 AlphaGo 原理剖析

提起增强学习和深度学习，现在最火热的项目非 AlphaGo 莫属。2016 年 3 月，AlphaGo 与韩国围棋棋手李世石进行了围棋的人机对弈，面对世界顶尖高手李世石，AlphaGo 最终以 4:1 的成绩战胜了李世石，改写了历史，更让全世界对人工智能的发展有了新的认识。AlphaGo 结合了机器博弈技术和深度学习技术，两者碰撞出的火花使得围棋这个人们心中的智能巅峰不再像过去那样高不可攀。下面我们将详细介绍 AlphaGo 背后的技术^[5]——机器博弈和深度学习模型。

27.6.1 围棋与机器博弈

围棋是源于中国的一项古老的棋类博弈项目，距今已有上千年的历史。它的游戏规则不算复杂，但是想要玩得好却不那么容易。围棋的核心规则可以用下面几句话概括出来：双方

各执一种颜色，在一个 19×19 的格子棋盘上放置自己的棋子，每一回合可以选择一个空着的位置放置一枚棋子。当一枚棋子或一组连通的棋子紧邻的所有位置都被对方的棋子包围时，这些棋子将被吃掉并拿出棋盘。最终棋盘上控制更多空间的一方获得胜利。

从博弈的角度来说，围棋是一个具有完备信息的博弈游戏，也就是说，双方的动作都将完全展示在对方面前，没有保留，这使得游戏的双方更容易做出正确的决策。围棋的游戏场景如图 27-6 所示。并不是每个博弈类游戏都拥有这样的特点，很多游戏会隐藏部分信息，使游戏更加复杂，并增加了很多不确定性。比如电子游戏 DOTA2，由于阴影的存在使得对战双方都很难掌握对方的全部动作信息，从而使战局充满了不确定性。如果 DOTA2 想做到和围棋一样具有完备信息，那么战场上就不应该有阴影存在，这样双方可以完全掌控对手的动作。DOTA2 的游戏场景如图 27-7 所示。

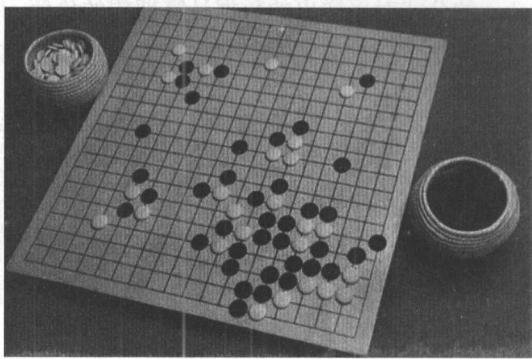


图 27-6 围棋游戏场景^[2]

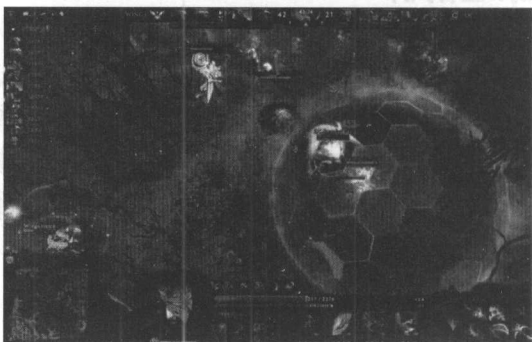


图 27-7 DOTA2 游戏场景^[4]

同时，围棋也是一个零和博弈的游戏，也就是说如果一方获得了胜利，那么另一方就将失败。用增强学习的方式来表示，对于一个状态 s ，就双方来说，两者的价值函数相加等于

0, 所以如果一方的价值函数为正数, 那么另一方的价值函数就为负数。这种对战型的博弈游戏普遍具有这样的特点, 上面提到的 DOTA2 同样具有这样的特点。但是, 和上面提到的例子有些不同, 围棋这样的博弈游戏是一种“交替”马尔可夫过程 (Alternating Markov), 当一方根据自己的策略完成动作之后, 另一方将同样采取自己的动作。而在 27.5 节提到的例子中, 采取策略的主体只有 Agent 一方, 环境的状态转移是有规律可循的。虽然我们不能简单地把之前的算法照搬到这里来, 但是问题的定义和之前十分类似。

(1) 对于对弈双方的其中一方来说, 这一方就是 Agent, 而棋盘和对手组成 Agent 所面对的环境 (Environment)。

(2) 每一轮棋盘上都会展示到目前为止的走子情况 (State), Agent 的目标是根据策略给出走子位置 a (Action)。

(3) Agent 完成走子后, 对手会根据 Agent 的走子情况给出走子招数, 这样棋局将转移到新的状态 (State)。与此同时, 环境还会将当前动作的奖励 (Reward) 发送给 Agent。对于奖励而言, 只有当游戏结束时, 双方才会获得实际的奖励: 获胜方获得 +1 的奖励, 失败方获得 -1 的奖励。在游戏过程中, 双方谁都不会获得奖励。

(4) Agent 将继续根据棋局给出走子位置, 直到棋局结束。

这个过程可以用图 27-8 来表示。

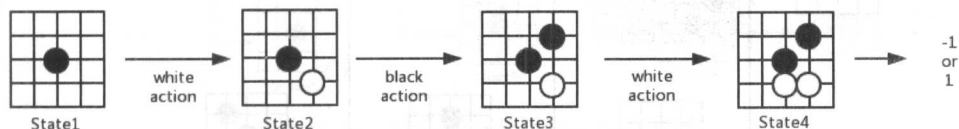


图 27-8 棋局演化图^[5]

由于无法得知对手的动作, 所以像之前的例子一样只考虑当前的状态是远远不够的, 还需要考虑自己走出招法后, 对手会采取什么样的应对措施; 针对对手的应对, 我方该采取什么样的后续招式。这样不断地思考下去, 就相当于对整个棋局做了推演。在各种博弈游戏中, 我们都会发现一个规律, 那就是有些在当前局势下看似不错的招法可能会为后面的招式埋下祸根, 而有些看似平淡的招式反而会为后面奠定良好的基础。在民间象棋界也曾有过“下棋看三步”这样的说法, 这也从侧面证明了推演招式的重要性。明确了推演招式的重要性后, 接下来我们就要看看如何推演。对于围棋来说, 完全的棋局推演是不可能的。所谓完全推演, 是指从当前的棋局出发, 对所有可能的招式进行分析, 并且一直分析到棋局结束, 然后根据每种招式的最终结果进行统计, 汇总并找出最优的招式。如果我们把推演的过程想象成一棵树, 那么当前的棋局就是树根, 每一种可行的招式就是一个树枝, 最终的结果就是一片叶子, 推演的过程相当于对可能出现的棋局的搜索过程。完整地生成这样一棵树需要多大代价呢?

对于围棋来说，这是一个难以想象的数字，据统计，围棋的棋局状态大概有 200^{180} 种，对这么多的状态进行枚举分析，对于现在计算机来说也是一个难以想象的负担。所以说完全推演全部过程是不可能的。

27.6.2 Alpha-Beta 树

既然完全推演不可能，那么部分推演可不可能？这个是可能的。所谓部分推演，就是只对这棵树的一部分进行搜索，或者说在搜索这棵树的过程中进行剪枝，忽略一些不重要的部分。剪枝可以从两个角度进行。首先，剪枝可以从每一层树的宽度开始。虽然每一步有很多可行的招式，但是实际上真正有价值的招式还是不多的。如果能够通过某种方法判断出哪些招式是有价值的，那么就可以减小树的宽度，为减少计算量做出第一步贡献，如图 27-9 所示。

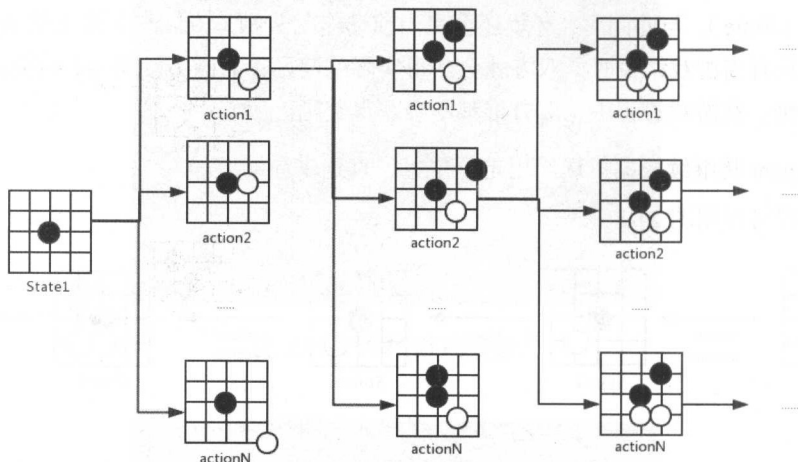


图 27-9 减小宽度的 Alpha-Beta 树^[5]

既然可以减小树的宽度，那么同样可以减小树的深度。在招法的推演过程中，不需要一直推演到棋局结束，而是可以在推演中间的某一步停下来，用某种方法判断在当前局势下哪一方优势大，并将这些优势信息汇总，同样可以计算出每一种招法的优劣，如图 27-10 所示。

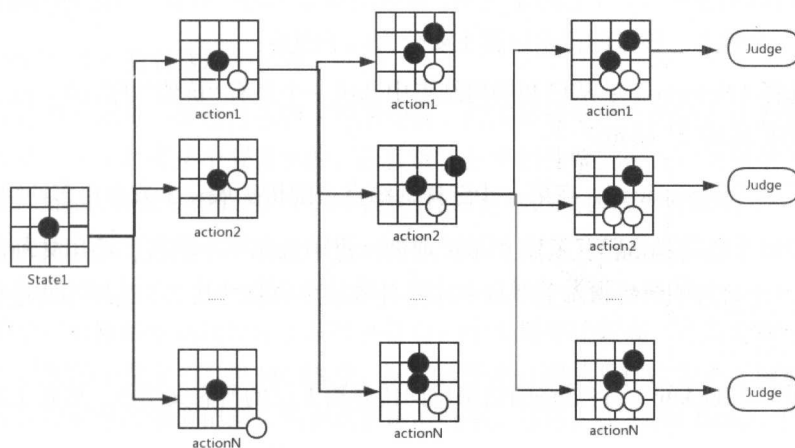
这两种方法可以有效地减少棋局推演的计算量，我们可以调节每个棋局推演的招数个数，也就是搜索树的宽度，以及总体搜索的招法数，也就是搜索树的深度。这样，棋局推演的计算量就可以控制在一个可以接受的范围内。我们可以简单地将上面的思路整理成伪代码：

```
def alpha_beta(board, depth, min, max):
    if depth > N:
        return evaluation(board)
```

```

moves=choose_n_move(board)
for move in moves:
    do_move(board)
    score = -alpha_beta(board, depth+1, -max, -min)
    undo_move(board)
    min = max(min, score)
    if min >= max:
        break
return min

```

图 27-10 减小深度的 Alpha-Beta 树^[5]

上面的代码就是机器博弈中经典的 Alpha-Beta 搜索树算法。由于对弈双方互为对手，对一方有利的招数对另一方就是不利的，所以在棋局判定时的回报分数在传向上一招汇总时会将所评估的分数取负数，这就是所谓的负极大值算法（Negative Max）。算法已经给出，但这个算法还存在两个问题，那就是选择有意义招法的“某种方法”和判断某个局面哪一方有优势的“某种方法”，而这两个算法与 AlphaGo 中的策略网络和价值网络类似。

27.6.3 MCTS

除了 Alpha-Beta 树，常见的机器博弈算法还有另外一种形式——蒙特卡罗树搜索（Monte Carlo Tree Search, MCTS）。蒙特卡罗树搜索是一种通过采样的方式近似求得目标期望的方法，对于一些难以求解的问题，采用大量随机采样的方式逼近目标值的方法就变得简单、粗

暴。有关蒙特卡罗方法的一个经典例子，就是求解 π 的具体值。而这种方法同样可以用在博弈中，尤其是像围棋这种难以分析表达模型的问题。之前我们提到，因为计算量太大，遍历所有的招法直至游戏结束这种方案是无法做到的。如果像 Alpha-Beta 树那样，当搜索进行到某一深度时进行局面判定，那么局面判定的算法就一定要做好。实际上，对于围棋这样复杂的棋局来说，局面判定的算法相当复杂，一般的模型很难能做好。既然如此，那么我们能不能暂时停止评估局面的想法，转而采用模拟走棋的方式判断最终结果呢？从结果上看，这个方案是可行的。只要不断地随机模拟后续招式，蒙特卡罗方法产生的结果就可以逼近我们最终想要的结果。一般来说，在大规模随机模拟棋局时，棋局有优势的一方获胜的局数会更多一些。在很多围棋博弈的算法中都采用了蒙特卡罗树搜索的方法。在蒙特卡罗树搜索中，每一个棋局都可以用树中的一个状态来表示，对弈的双方走出一招后，当前的棋局将从一个状态转移到另一个状态。核心的搜索过程主要分为以下四步：

(1) 选择 (Selection)：从已知的树结点中选出一个最有“价值”的结点，这个价值通过结点的“急迫扩展程度”决定。

(2) 扩展 (Expansion)：在第一步选择的结点上随机扩展一个或者更多的结点。

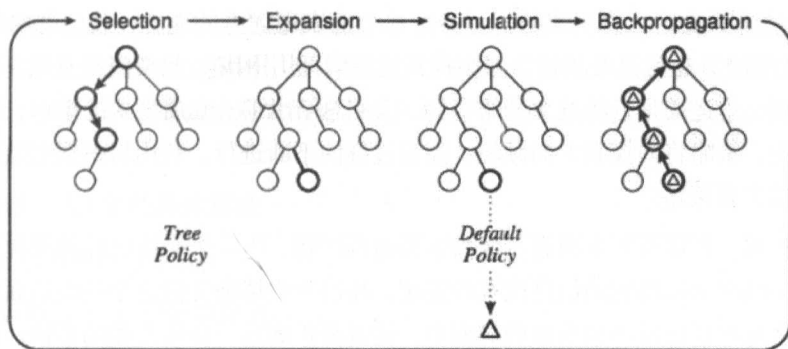
(3) 模拟 (Simulation)：从第二步扩展的结点中选择一个结点，采用某种策略随机生成新的状态，并在这个新的状态上继续利用这种策略生成新的状态，这样不断进行下去直到最终状态。

(4) 回传 (Backup)：将最终的结果反向传导给上层的所有父结点，更新父结点的相应信息。

其伪代码如下：

```
function MCTSSEARCH( $s_0$ )
    根据状态  $s_0$  创建根结点  $v_0$ 
    while 停止条件未满足 do
         $v_l \leftarrow \text{TREEPOLICY}(v_0)$ 
         $\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$ 
        BACKUP( $v_l, \Delta$ )
    return a(BESTCHILD( $v_0$ ))
```

对应的流程如图 27-11 所示。

图 27-11 MCTS 算法流程图^[5]

下面我们具体来看看这四个步骤。

首先是选择。在每一次进行蒙特卡罗演绎前，都需要选定一个树结点作为蒙特卡罗推演过程的出发点。这个过程是十分重要的，而且蒙特卡罗树搜索的过程和增强学习中的经典问题——Multi-Bandit 有很多相似性。两个问题同样对过程的整体次数有限制：在 Multi-Bandit 中玩家操纵老虎机摇杆臂的次数是一定的，而在 MCTS 中采样的总次数是固定的，因为这涉及程序的运行时间。另外，每一次流程结束，两个问题都会获得一定的信息，在 Multi-Bandit 中，玩家希望在有限的游戏次数中了解哪些摇臂的中奖概率比较高，并且能够尽可能精确地计算出这些摇臂的中奖率；而在 MCTS 中，玩家希望在有限的采样次数中了解哪些招式是高质量的，并且能够尽可能地记录这些招式精确的胜率。这样，在 Multi-Bandit 中遇到的问题，在 MCTS 中也有可能遇上，而这个问题也真的遇上了，那就是探索-利用问题（Exploration-Exploitation）。所谓的探索，是指尝试之前没有用过的一个动作情况，如果能找到一些更好的动作，这将会对最终的策略起到更大的帮助作用。对于 Multi-Bandit 问题，我们需要尝试一些没有操纵过的摇臂和之前中奖率低，因为没有操纵过的摇臂可能中奖率很高的摇臂，而之前中奖率低的摇臂可能是因为运气差，而实际上这些摇臂的中奖率很高；对于 MCTS 问题，我们希望多尝试一些不同的招式和之前判定胜率低的招式，因为没有出现过的招式也许才是胜负的关键，而之前胜率低的招式也可能是因为随机采样的次数不够，一旦采样次数多到能够在一定程度上反映其真实胜率时，我们就会发现它们其实是胜率很高的招式。所谓的利用，就是指利用当前已知的情况实施策略。对于 Multi-Bandit 问题，如果已经知道某个摇臂的中奖率很高，那么我们会尽可能地把剩下的摇臂次数用在这里，因为它会更大概率地为我们获得收益；对于 MCTS 问题，如果已知一个招式的胜率很高，那么为了进一步明确这个招式是否真的胜率很高，我们就需要把有限的采样次数尽可能地用在这个局面上，这可以帮助我们更加确定这个招式的胜率。无论是 Multi-Bandit 还是 MCTS 问题，从这两个问题来看，探索和利用似乎是一对矛盾体，因为总体的资源有限，无论是 Multi-Bandit 的摇臂次数

还是 MCTS 的采样次数，将更多的资源分配给当前表现优秀的地方，就会忽略掉一些潜力股；反之也是如此。这就是在增强学习中提到的探索-利用困境。只要资源有限，我们就无法摆脱这个困境。幸运的是，经过大量的实验，人们总结出了一套启发式的策略，那就是在整个过程的初期，策略将向探索方向倾斜，随着过程的不断进行，利用将占据主导地位。这也是选择过程的主要策略。

然后看扩展。扩展看上去则有点像探索策略的产物，当一个局面已经被采样到一定程度后，再从这个局面采样得到的信息就有些笼统，我们希望精确分析这个局面后面的一些具体信息，从而将这些信息反馈到前面的局面中，这就是扩展这一步所要做的事情。这一步的关键在于何时进行扩展，以及如何扩展。

接下来看模拟。这是蒙特卡罗方法的真正过程。在这个过程中，算法将采用模拟的方式代替双方走子，从上一步选定的局面开始，一直走到结束，并记录最终的结果。一般来说，为了保证双方的出招不至于太离谱，招式的概率分布并不是一个均匀分布，这里同样需要一个招法模型为每种招式计算出现的概率。只不过这里的模型并不需要特别精确，如果这个模型能做到排除绝大多数没有意义且在实战中绝对不会出现的招式，那么它就是一个好的模型。

最后看回传，也就是将采样模拟的结果回传到起始的局面。这里的回传还涉及一些细节过程。

27.6.4 UCT

在众多算法中，比较有名的算法之一就是 UCT (Upper Confidence Bound applied to Trees)，是蒙特卡罗搜索方法和 UCB (Upper Confidence Bound) 公式的结合。

首先看 UCT 在选择部分的判别公式，这也是它最具特色的地方。它在选择过程中的判别算法 UCB1 为：

$$UCT = \overline{X_j} + 2C_p \sqrt{\frac{2 \ln n}{n_j}}$$

其中 $\overline{X_j}$ 表示这个结点当前所有蒙特卡罗过程后的平均得分，这象征了该结点的招式经过蒙特卡罗演绎后的效果； C_p 是一个参数，用于调整模型对于探索和利用之间的权重关系； n 为当前已经进行的蒙特卡罗的轮数； n_j 表示该结点已经进行的蒙特卡罗的轮数。从这个公式可以看出，一个结点被选中，要么是因为它最终获胜的概率高，也就是公式右边的第一项大；要么是因为它被随机的次数少，这样公式右边的第二项就会大些。这个公式并不复杂，但是在实际应用中效果很不错，所以被广泛采用。除此之外，UCT 中的其他函数和 MCTS 中的内容相似。树中的每一个结点都保存了两个值：访问该结点的次数和累计的奖励数量。将

两个数字相除就可以得到这个结点的平均奖励，也就是期望胜率。对于像围棋这样的零和博弈问题，在奖励结果回传的过程中，同样需要对不同层的奖励取反，如果结果是我方胜利，且给我方的记分是 1 分，那么给对方的记分就应该是 0 分或者 -1 分。基于上面的分析，下面给出 UCT 的具体算法。

算法 27-8 UCT 的具体算法

function UCTSEARCH(s_0)

依据状态 s_0 构建根结点 v_0

while 停止条件未满足 **do**

$v_l \leftarrow \text{TREEPOLICY}(v_0)$

$\Delta \leftarrow \text{DEFAULTPOLICY}(s(v_l))$

BACKUP(v_l, Δ)

return $a(\text{BESTCHILD}(v_0, 0))$

function TREEPOLICY(v)

while 非叶子结点 v **do**

if v 没有被完全扩展 **then**

return EXPAND(v)

else

$v \leftarrow \text{BESTCHILD}(v, C_p)$

return v

function EXPAND(v)

选择 $a \in A(s(v))$ 中未尝试的动作

增加一个 v 的新子结点 v' ，并令 $s(v') = f(s(v), a)$, $a(v') = a$

return v'

function BESTCHILD(v, c)

return $\arg \max_{v' \in (\text{children of } v)} \frac{Q(v')}{N(v')} + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$

function DEFAULTPOLICY(s)

while s 是非叶子结点 **do**

随机选择 $a \in A(s)$

$s \leftarrow f(s, a)$

```

return 状态  $s$  的奖励

function BACKUP( $v, \Delta$ )
    while  $v$  is not null do
         $N(v) \leftarrow N(v) + 1$ 
         $Q(v) \leftarrow Q(v) + \Delta(v, p)$ 
         $v \leftarrow v$  的父结点

```

AlphaGo 在实际中使用了 UCT 的一个变种——PUCT 算法，在后面我们会详细介绍这个算法。

至此，我们已经详细介绍了与围棋相关的两个机器博弈算法——Alpha-Beta 树和 MCTS，对于 MCTS，详细介绍了每一个步骤和其中的细节。下面就来看看 AlphaGo 中涉及的模型和 AlphaGo 的搜索策略。

27.6.5 AlphaGo 的训练策略

上面介绍了针对围棋机器博弈的两种方法：传统的 Alpha-Beta 树形式的方法，通过策略网络和价值网络的方式求解最优招式；利用 MCTS 的方式求解最优招式。这两种方法虽然不同，但是都用到了策略网络这个模型。不过，这两种方法对模型的要求不同。对于 Alpha-Beta 树的方式来说，策略网络主要用于从树的宽度方向剪枝，因此这个模型的精度对最终的效果很重要；而对于 MCTS 的方式来说，策略网络主要用在蒙特卡罗棋局演绎的地方，这里需要在保证招式基本合理的基础上尽可能快地完成。所以，实际上这两种方法对策略网络的需求是不同的。

在早期的机器博弈中，出招算法和局面评估算法都是采用规则的方法。作者曾经研究过六子棋的机器博弈问题，所谓的六子棋和五子棋类似，其最终目标是为了使六枚棋子连成一条直线或斜线。在这个问题中，作者曾经采用枚举各种棋形的方式来分析出招和局面评估的优劣。六子棋除先手第一式只走一枚棋子外，其余每一手都会走两手。如果走出某一招式后棋子已经形成四子连续或者五子连续，那么对方将被迫做防守，这样的招式就容易被选出。另外，如果对方形成了四子连续或者五子连续，那么我方将其进行截断或者封锁，同样会因为防守而被选中。这样，问题就变成了对棋局上棋形模式的判定以及对未来棋形走向的分析。这种基于规则的做法有两个好处：一是可以根据规则快速地实现一个具有一定棋力的博弈系统；二是出招的算法和局面评估都具有一定的可解释性。当然，这种方法也有两个弱点：一是需要模型的设计者具有一定的棋力，随着模型的能力不断提高，设计者的棋力也需要不

断提高,才能设计出更加精妙的模型;二是对于复杂的棋类游戏,规则会变得越来越复杂,以至于调节的代价越来越大。基于上面两个问题,一些模型设计者转而采用机器学习的方式进行模型设计。所采用的模型以浅层网络为主,这些模型在一些相对简单的棋类问题上取得了良好的效果,而在一些复杂的棋类问题上却不太容易产生好的效果,比如围棋。由于围棋需要考虑的问题十分抽象,采用 Logistic Regression, SVM 等模型进行出招和局面评估的判定,都需要特征工程对落子点或者局面进行特征抽取,而这仍然是一个十分困难且精度不高的问题。所以,关于围棋的模型一直处于一个瓶颈期,无法有质的提升。当然,今天的我们已经知道,AlphaGo 利用深度学习的方法战胜了世界一流选手,完成了质的突破并震惊了世界。下面就来看看 AlphaGo 是如何把这两个模型训练出来的。关于 AlphaGo 技术细节的论文中介绍了在训练过程中实际上产生了四个模型:

- 利用监督学习训练监督策略网络 (Supervised Policy Network) 和快速策略网络 (Fast Policy Network)。
- 利用增强学习训练增强策略网络 (Reinforcement Policy Network)。
- 利用监督学习训练价值网络 (Value Network)。

这四个模型的关系如图 27-12 和图 27-13 所示。

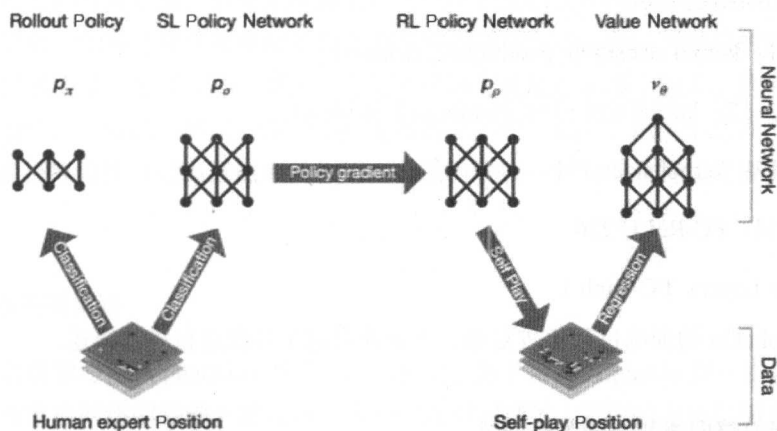
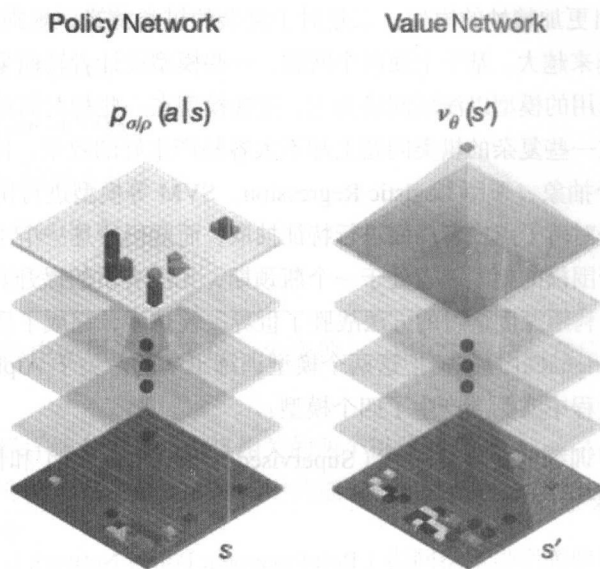


图 27-12 AlphaGo 四种网络关系图^[5]

Policy 的网络结构如下:

- Layer1: kernel size=5*5, padding=2, stride=1
- Layer2~12: kernel size=3*3, padding=1, stride=1
- Layer13: kernel size=1*1

图 27-13 AlphaGo 两种网络结构图^[5]

Value 的网络结构如下：

- Layer1: kernel size=5*5, padding=2, stride=1
- Layer2~12: kernel size=3*3, padding=2, stride=1
- Layer13: kernel size=1*1
- Layer14: FC-ReLU 256
- Output Layer: FC-Tanh 1

由于 AlphaGo 的训练过程比较复杂，下面就对这个过程进行详细描述。

1. 训练监督策略网络和快速策略网络

这一步是从 0 开始的，为了让 AlphaGo 从一个小白快速成长，学习别人的棋谱就成了一件十分重要的事情，这就是监督学习的任务。AlphaGo 的研发团队首先收集到了一批高水平棋手对决的棋局，将这些棋局中的每一招拆解成对应的数据对，并根据棋局信息对每一个落子点进行了特征提取，然后利用这样的训练数据对出招模型进行训练。那么训练数据有多少呢？官方给出的数量是 3000 万。这个数据量不是个小数目，不过要训练的模型也是一个 13 层的卷积神经网络，其中包含的参数非常多，复杂度也非常高，所以相比而言这个数据量也

就没那么多了。对于每一个走子位置，AlphaGo 抽取了 48 个特征。这些特征大部分是落子点附近的一些简单的信息，比如当前位置附近是否有空位等；也有一些相对复杂的特征，例如动态的空位信息；还有棋局本身的特征，用于记录落子的合法性（打劫）和是否会自杀。这些特征都可以帮助模型做更好的判断。这批数据同时训练出了两个模型，一个是神经网络层数相对比较深的监督策略网络，这个网络的精度比较高，但是速度相对较慢；另一个是层数相对比较浅的快速策略网络，这个网络的精度比较低，但是速度极快。它们的准确率和运行时间如表 27-1 所示。

表 27-1 两种策略网络的性能

	监督策略网络	快速策略网络
准确率	57%	24.2%
运行时间	3ms	2 μ s

从表 27-1 中可以看出，两者的准确率的差距比较大，运行时间的差距也很大。考虑到训练准确率低可能的两个原因——棋局本身招式存在问题 and 某个棋局存在相同效果的招式，上面相差比较大的准确率并不能说明两个模型的效果存在巨大的差距。首先，所有的训练数据都来自于线上的对弈数据，棋局是由高水平的棋手完成的，本身可以保证落子的质量，但是并不能保证每一招都是最优策略而不存在有某些问题的招式。所以与棋局的落子结果不同并不能说明模型的结果是错误的；其次，在某个棋局下可能存在多个妙手，这种情况经常出现在棋局开始阶段，所以与棋局的落子不同也可能是妙招。此外，论文中还列出了其他一些数据，如验证提取特征对提高准确率的作用。这说明添加一些简单的特征对提高准确率有很大的作用。

2. 训练增强策略网络

前面的监督学习使 AlphaGo 获得了一定的棋力，为了帮助 AlphaGo 进一步提高棋力，研发团队在监督策略网络的基础上进行进一步的增强学习训练。采用的方法是用当前训练的模型和早先的模型进行机器和机器之间的对弈，并将对弈的棋局整理成训练数据进行训练，最终得到了增强策略网络（Reinforcement Policy Network）这个模型。前面的训练已经产生了模型，拥有这些模型，可以让模型之间相互进行对弈，产生全新的训练数据。通过这种方法，又有大量的棋局数据被生成出来，这些数据便可以用来进行进一步的训练。如果有走得好的招式，则可以学习进行进一步的提高；如果有走得不好的招式，则可以学习进行纠正。在模拟对弈中，增强策略网络对监督策略网络有 80% 的胜率。这个模型最终没有进入 AlphaGo 对弈的模型中，但是却为后面模型的训练奠定了基础。

3. 利用监督学习训练价值网络

价值网络和策略网络的结构类似，主要不同的地方是它的输出只有一个值，作为对局面最终结果的评估。这个模型在曾经的围棋博弈系统中比较少见，因为传统的方法很难把这个问题做好，但是利用深度学习模型，价值网络的表现出乎意料地有了很大的提高。由于训练数据中已经包含了每一局棋的结果，因此直接抽出棋局中的某一个局面，配上最终结果，就构成了训练数据。在训练过程中，如果采用完整的局面进行训练，则会造成过拟合。因为相近的招式之间的棋局差距不大，而且对于一个完整的棋局来说，真正决定棋局胜负的关键往往出现在某几招中，不会贯穿于全部棋局，所以最终的训练方式是采样每个棋局中的招式，这样就极大地避免了过拟合的问题。

27.6.6 AlphaGo 的招式搜索算法

AlphaGo 的招式搜索算法结合了传统的 MCTS 算法和 Alpha-Beta 树算法，具体的不同之处在于 MCTS 的模拟部分，除了采用之前的蒙特卡罗模拟方法，还可以采用价值网络直接进行结果的预测。经过实验测试，同时结合蒙特卡罗方法和价值网络方法效果是最好的。除了这部分，每个结点还会拥有一个独立的招式分数，这个分数来自于监督策略网络。这样，所有的模型都很好地集成到同一个体系中，那么我们可以列出 AlphaGo 的算法流程如下。

- (1) 将当前的棋局初始化为根结点。
- (2) 选择：利用 PUCT 的选择算法选择一个结点。
- (3) 扩展：对所选择的结点扩展它的叶子结点，并采用监督学习策略网络对所有的结点计算其先验奖励概率。
- (4) 模拟：选择一个叶子结点，同时采用价值网络和快速策略网络计算最终的奖励。
- (5) 回传：将动作的价值向上传递，更新所有的父结点。
- (6) 最终选择奖励最高的结点作为所选定的招式。

下面我们来看看 AlphaGo 中 MCTS 的一些细节。首先看 PUCT，PUCT 的计算方法如下：

$$u(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N_r(s, b)}}{1 + N_r(s, a)}$$

其中 $P(s, a)$ 是当前状态-招式对的先验概率，这个概率由监督学习策略网络计算得到； c_{puct} 是一个常量，表示探索的程度。这个公式和 UCT 的不太一样。在 AlphaGo 的 MCTS 树

中，每一个树结点都保存了以下信息：

$$\{P(s, a), N_v(s, a), N_r(s, a), W_v(s, a), W_r(s, a), Q(s, a)\}$$

其中 $P(s, a)$ 是状态-招式对的先验概率； $N_v(s, a)$ 是采用价值网络计算的次数； $W_v(s, a)$ 是采用价值网络计算的动作价值； $N_r(s, a)$ 是采用 Rollout，也就是蒙特卡罗方法模拟的次数； $W_r(s, a)$ 是采用常规的蒙特卡罗方法计算的动作价值；而最后的 $Q(s, a)$ 是结合价值网络和蒙特卡罗两种方法计算的动作价值。在反向传递更新信息时，对于常规的蒙特卡罗方法，如果经过 n_r 轮模拟，某个招式赢得了 v_r 场胜利，那么更新的公式为：

$$N_r(s, a) += n_r$$

$$W_r(s, a) += v_r$$

价值网络的更新方法和常规的蒙特卡罗方法不太一样，由于价值网络会产生 0 到 1 之间的结果 v_v ，所以从数值上看它相当于一次蒙特卡罗方法的计算，所以它的更新公式为：

$$N_v(s, a) \leftarrow N_v(s, a) + 1$$

$$W_v(s, a) \leftarrow W_v(s, a) + v_v$$

对最终两部分的结果进行加权平均，我们定义常规的蒙特卡罗方法的权重为 λ ，那么 $Q(s, a)$ 的计算公式为：

$$Q(s, a) = (1 - \lambda)(W_v(s, a)/N_v(s, a)) + \lambda(W_r(s, a)/N_r(s, a))$$

从上面的算法可以看出，三个核心模型相互弥补，组合起来更像一个 Ensemble 的方案。后面的实验对这三个模型的各种组合方式进行了测试，并验证了三个模型结合的效果。最终的实验结果表明，三个模型同时启用效果最好，且两种方法的权重相同时效果最好，也就是 $\lambda=0.5$ 。

此外，在进行搜索时，AlphaGo 还采用了异步的分布式搜索方法，以保证在短时间内能够完成大规模的计算。这个分布式算法采用 Master-Worker 的架构进行。其中一台 Master 机器负责 MCTS 的主体部分，大量的 CPU Worker 完成蒙特卡罗 Rollout 的部分，大量的 GPU 完成策略网络和价值网络的计算工作。由于其中涉及一些信息的同步，因此这个分布式算法还是具有一定的复杂性的。

27.6.7 围棋的对称性

最后我们再来谈谈 AlphaGo 中的细节问题。由于围棋的棋盘是对称的，在预测过程中互为对称的几个棋盘在理论上应该得到相同的结果。对于这个问题，不同的模型有不同的解决方法。有的围棋博弈系统在模型中加入了解决围棋对称性的模型参数；而 AlphaGo 则直接将棋盘镜像复制 8 份，将 8 个棋盘同时送入模型中进行计算，然后将 8 个棋盘的数值进行平均，就可以求出最终的结果。采用这样的方法可以使模型更专注于原始的任务，不需要考虑镜像棋盘这样的问题。而且，并发求解对于分布式系统来说在速度上不会有太大的劣势。

至此，我们就完成了对 AlphaGo 的技术介绍。可以看出，AlphaGo 的整体架构和之前的围棋人工智能相比并没有太大的区别，但是它很好地将深度学习和增强学习结合进来，使得其模型结构比之前的人工智能更加复杂，拥有的训练数据也更多。再加上这些年不断提升的计算速度，多方面的因素成就了 AlphaGo 今天的成功。围棋人工智能的成功预示着计算机已经基本可以在棋类博弈上超越人类，但是人类还有很多方向是远胜计算机的，计算机的征途只完成了一小部分，更大的未来、更多的精彩还在等着我们去发现、探索。

27.7 AlphaGo Zero

2017 年 10 月 19 日，Google DeepMind 的 Nature 论文 *Mastering the game of Go without human knowledge* 公开了新版围棋程序 AlphaGo Zero，仅仅使用 4 个 TPU，无需任何人类经验，从零开始左右互搏训练 3 天，共计 490 万局，就以 100:0 击败了之前与李世石对弈的 AlphaGo Lee。

论文中提及 AlphaGo Zero 与 AlphaGo Lee 的最大区别仅为如下四点：

- (1) 不需要任何人类经验，从零起步自我训练。
- (2) 只使用棋盘上的黑白棋作为输入特征。
- (3) 不再独立使用策略网络和价值网络，统一为单个网络。
- (4) 在单个网络上使用更简单的树搜索来评估位置和招法，抛弃了原来的蒙特卡罗树搜索中的 Rollout 策略，直接依靠深度学习网络输出来做落子位置选择和评估。

参考文献

- [1] Sutton R S, Barto A G. Reinforcement learning: an introduction[J]. Neural Networks IEEE

Transactions on, 2013, 9(5):1054.

[2] Browne C B, Powley E, Whitehouse D, et al. A Survey of Monte Carlo Tree Search Methods[J]. IEEE Transactions on Computational Intelligence & Ai in Games, 2012, 4:1(1):1-43.

[3] Mnih V, Kavukcuoglu K, Silver D, et al. Playing Atari with Deep Reinforcement Learning[J]. Computer Science, 2013.

[4] Atari 2600-百度百科 http://baike.baidu.com/link?url=-MAcoQAb_GKZZlnIv-s7ylwxA7ljC887yIfC-OyxlyOw2kgF6_U2m5d8QQeVH_BH7mYTswXnWraO8kMCDcCaOxDOau9573m2p7OHTgxfg3u.

[5] Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search.[J]. Nature, 2016, 529(7587):484.

[6] If Pac Man for the Atari 2600 Was Released Today. <http://mangotron.com/if-pac-man-for-the-atari-2600-was-released-today/>.

28

GAN

前面介绍了 RBM、DBN、自动编码器等用于初始化的生成模型，也介绍了 CNN、RNN 等判别模型，本章将介绍一种新的生成模型框架——生成对抗模型（Generative Adversarial Network, GAN）^[1]，在这种模型框架中既有生成模型也有判别模型。

28.1 生成模型

生成模型主要用于描述数据分布的情况，构建生成模型可以满足数据采样等不同方面的需求。对于简单的数据分布的描述方法，相信读者已经有所了解了。比如，我们可以假设数据满足某种分布，如高斯分布，这样数据分布的形式就已经基本确定，接下来的任务是求解分布中的参数。对于高斯分布，要求解其中的均值和方差，这样我们就拥有一种对数据描述的形式。构建的过程如图 28-1 和图 28-2 所示。

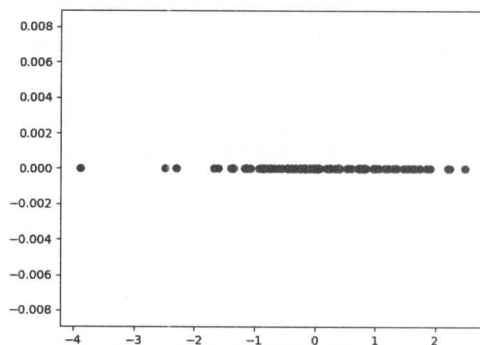


图 28-1 简单的数据集

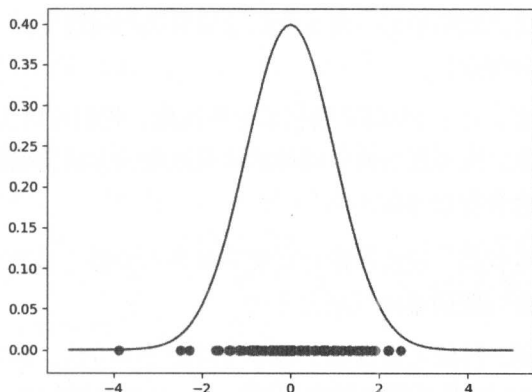


图 28-2 利用高斯分布拟合的模型形式，其中纵轴表示数据出现的概率

对于简单的问题，采用这样的方法就可以解决。但是如果数据的形式复杂，利用常见的分布进行描述便显得有点吃力，比如图 28-3 所示的数据。

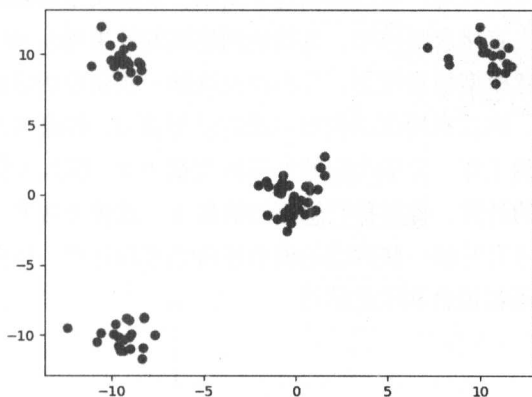


图 28-3 有点复杂的数据

对于这些数据，一个直观的感觉是——数据大体被分成了 4 个部分，如果用 4 个简单的分布来表示，是不是就可以了呢？当然是可以的，但是由于这些数据来自于一个整体，那么在描述时一定要明确指出数据是从这 4 个部分中的哪一个部分产生的。像这样的描述虽然不会出现在数据的表示中，但是对数据的产生却起到了十分重要的作用，这种变量一般被称作隐含变量。

如果用概率建模的方式来表述，那么对于图 28-1 和图 28-2 所示的数据，我们用 $P(X)$ 的形式进行建模，其中 X 代表数据；而对于图 28-3 所示的数据，我们就可以用 $P(X|Z)$ 的形式进行建模，其中 Z 表示隐含变量，在这个问题中，也就是“数据属于哪一个部分”。当然，还需要对 $P(Z)$ 进行建模。

实际上,上问题还是比较简单的,现在我们希望解决的问题是描述某类图像的分布,这类数据有如下两个明显的特点。

- 数据的维度比较高,对于 MNIST 数据集中的图像,我们可以认为每张图都是 28×28 维空间中的一个点,这样复杂的空间很难用可视化的方法精确描绘出来,因此就不能用上面的方法将隐含变量分析出来。
- 数据间的相关程度较高,导致数据的真实分布并不能够充满整个空间,往往只存在于子空间内,这使得问题的难度又增大不少。

为了解决这样的问题,科研人员发明了很多为其建模的方法。其中一类方法和上面建模的思路类似,但不同的是,模型的复杂程度相对较高。有些模型包含了很多变量、很多层次,虽然比较容易求解,但是相比数据的形式还是不够复杂;有的模型足够复杂,但是却不易求解,为了效率考虑只能选择近似求解。总之,为了求解出数据分布的具体形式,在建模的过程中充满了困难。

这时又有一些科研人员在想,既然数据过于复杂,求解模型的具体形式比较复杂,那么能不能换一个思路,不去直接求解模型,也能达到建模的效果呢?利用深度学习的模型,这个想法变成了现实。既然模型很难建立,我们就先选择一些简单的数据进行构建,得到一个“隐含变量”的数据分布,然后利用深层模型超强的映射能力,将隐含变量和观察数据对应起来,这样就完成了最终的工作。令那个起始的随机变量为 z ,那么 z 将从某个分布 $p(z)$ 中采样出来,然后经过模型的计算,得到我们想要的数 x 。这种方法有一个好处,就是不需要再进行复杂的建模,况且有时候一些构建的模型还特别难以计算。现在拥有了可以求导的深层模型,我们只要完成函数拟合和优化即可。

28.2 生成对抗模型的概念

生成对抗模型^[1]的建模思想是基于上一节结尾描述的理念的。除此之外,它还采用一种博弈的方式进行训练。在生成对抗模型中,我们将要同时训练两个模型:一个是生成模型 G ,也就是靠数据拟合生成目标数据的模型;另一个是判别模型 D ,这个模型将作为生成模型对手,站在它的对立面进行训练。前面已经介绍过生成模型的结构,而判别模型的形式和我们常见的判别模型很类似,它的输入是原始数据或者生成模型生成的数据;输出是一个概率值,代表数据来自于原始数据的概率。

之所以说两个模型是对立的,是因为两者的目标不同。生成模型 G 的目标是希望自己生成的数据被判别模型判定为“来自于原始数据”;而判别模型 D 则希望能把原始数据和生成数据区分开来。基于这样的目标,我们可以给出这对模型的目标函数:

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

由于这个函数涉及两个模型，而且模型的目标不同，所以我们采用分开训练的方式进行训练。首先训练判别模型，让它拥有一定的判别能力，然后再训练生成模型，让它基于判别模型的能力提升自己的能力。这样生成模型就像一个制作假冒商品的骗子，判别模型就像一个检查假冒商品的警察，警察提高了侦查能力，骗子想要继续行骗就需要提高自己的骗术。这样一来一回，两方面的能力就都得到了提高。

除了这个模型的理念类似“魔高一尺，道高一丈”，它的训练方式也与此十分类似。在训练过程中也同样采用交替训练优化的方式。首先训练判别模型，然后训练生成模型，接下来再训练判别模型，然后又训练生成模型……如此不断地进行训练，直到生成模型达到想要的效果。GAN 训练的模型结构如图 28-4 所示。

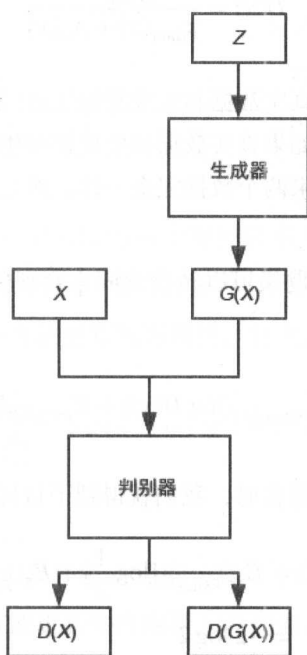


图 28-4 GAN 模型结构图

由于生成模型和判别模型出现在同一个目标函数中，并且算法采用了交替更新的方式，因此这里需要对算法的内容进行分析——分析两个模型在交替优化过程中的表现。首先生成

模型被固定，那么判别模型的目标函数将变为：

$$\begin{aligned}
 \max_D V(D, G) &= E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(x)}[\log(1 - D(G(z)))] \\
 &= E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{x \sim p_G(x)}[\log(1 - D(x))] \\
 &= \int_x p_{\text{data}}(x) \log D(x) + p_G(x) [\log(1 - D(x))] dx
 \end{aligned}$$

如果希望这个目标函数在理想状况下最大化，那么可以让判别函数在每一个点上都取得最大值，于是就有：

$$d(x) = p_{\text{data}}(x) \log D(x) + p_G(x) [\log(1 - D(x))]$$

这里让上面的公式取得最大值，就有：

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}$$

从这个公式可以看出，如果真实数据和生成模型生成的数据差距很大，那么经过充分训练的判别模型就会将两者分开。如果真实数据和生成模型生成的数据差距很小，那么判别模型将很难发现它们的区别。而如果两个数据完全一样，那么判别模型对于任意一个数据的概率都为 50%。

了解了判别模型的内容，问题又可以转变为固定判别模型，求解生成模型的目标函数。我们可以重新转换目标函数：

$$\min_G V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(x)}[\log(1 - D(G(z)))]$$

当生成模型和真实数据完全重合时，我们就得到了目标函数的最小值：

$$\begin{aligned}
 \min_G V(D, G) &= E_{x \sim p_{\text{data}}(x)}[\log \frac{1}{2}] + E_{z \sim p_z(x)}[\log(\frac{1}{2})] \\
 &= -2 \log 2
 \end{aligned}$$

而如果考虑在迭代过程中某一轮迭代都生成模型的优化目标，就有：

$$\begin{aligned}
\min_G V(D, G) &= E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{x \sim p_G(x)}[\log(1 - D(x))] \\
&= E_{x \sim p_{\text{data}}(x)}\left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}\right] + E_{x \sim p_G(x)}\left[\log \frac{p_G(x)}{p_{\text{data}}(x) + p_G(x)}\right] \\
&= E_{x \sim p_{\text{data}}(x)}\left[\log \frac{p_{\text{data}}(x)}{\frac{p_{\text{data}}(x) + p_G(x)}{2}} - \log 2\right] + E_{x \sim p_G(x)}\left[\log \frac{p_G(x)}{\frac{p_{\text{data}}(x) + p_G(x)}{2}} - \log 2\right] \\
&= -2 \log 2 + \text{KL}(p_{\text{data}}(x) \parallel \frac{p_{\text{data}}(x) + p_G(x)}{2}) + \text{KL}(p_G(x) \parallel \frac{p_{\text{data}}(x) + p_G(x)}{2})
\end{aligned}$$

从公式中可以看出,生成模型的目标就是不断地让自己的数据分布和真实数据的分布靠近。其中右边的两个 KL 项又可以组成一个新的散度度量——Jason-Shannon 散度(以下简称 JSD)。这个散度拥有和 KL 散度类似的性质,可以测量两个分布的差异。两者最大的不同在于 JSD 是一种对称的度量方式,也就是说, $\text{JSD}(A \parallel B) = \text{JSD}(B \parallel A)$, 因此不会出现在 KL 散度中因为不对称造成的问题。

GAN 的总体算法如下:

算法 28-1 GAN 总体算法

for $T = 1$ to 训练迭代轮数 **do**

for k 轮迭代 **do**

 从噪声先验分布 $p_g(z)$ 中采样出 m 个噪声样本 $\{z^{(1)}, \dots, z^{(m)}\}$ 作为一批训练数据

 从真实数据中采样 m 个样本 $\{x^{(1)}, \dots, x^{(m)}\}$ 作为一批训练数据

 利用随机梯度上升的方法更新判别模型, 公式为:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))]$$

end for

 从噪声先验分布 $p_g(z)$ 中采样出 m 个噪声样本 $\{z^{(1)}, \dots, z^{(m)}\}$ 作为一批训练数据

 利用随机梯度下降的方法更新生成模型, 公式为:

$$\nabla_g \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(z^{(i)})))$$

end for

28.3 GAN 实战

在实战过程中，GAN 模型还有一些需要注意的问题。由于它的优化方式和往常的模型不同，两个模型之间存在着竞争关系，所以在优化过程中还是需要注意很多问题的（后面介绍的方法会一步步解决这些问题）。

首先，由于生成模型在反向计算时要经过判别模型，所以如果目标函数对判别模型特别有利的话，梯度在经过目标函数时就会消失。从上面介绍可以看出：

$$\min_G V(D, G) = E_{z \sim p_z(x)} [\log(1 - D(G(z)))]$$

经过计算我们会知道，如果判别模型的能力很强，可以完美区分生成模型的图像和真实图像，那么生成模型的梯度就会消失——因为对于判别模型来说，优化已经没有梯度了。为此，目标函数可以做一定的调整：

$$\min_G V(D, G) = -E_{z \sim p_z(x)} [\log(D(G(z)))]$$

这样即使判别模型可以完美区分，生成模型也可以获得想要的梯度。

很多研究人员对基于 CNN 的 GAN 模型结构做了研究，其中 *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*^[2] 中提到了有关模型结构的建议：

- 将池化层做替换，判别模型可以将其替换为 Strided Convolution，生成模型可以将其替换为 Fractional Convolution。
- 在生成模型和判别模型中使用 Batch Normalization 层。
- 对于比较深的结构，去掉全连接层。
- 对于激活函数的使用，在生成模型中使用 ReLU，最后一层输出使用 Tanh；对于判别模型，使用 Leaky ReLU，不要使用 Sigmoid。

除此之外，两个网络之间的竞争学习还需要特别注意。由于模型间的竞争关系，两个模型都不能比对方强大太多，如果判别模型强大太多，生成模型就无法快速成长；如果生成模型强大太多，一旦找到一些“窍门”，它将“不思进取”，尽可能地利用这些窍门来获得较低的 loss。为此，*Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* 中还提到了有关这一部分的建议：

- 让判别模型使用一个和生成模型相比小一些的模型。因为生成模型需要生成一幅图像，难度相对比较大，所以需要更多的参数也是理所应当的；而判别模型只需要给出一个结果，难度相对较小，所以使用一个较小的模型更合适一些。
- 在判别模型中使用 Dropout，这样会使判别模型不易过拟合，因而不容易被生成模型的一些奇怪的图像所迷惑。
- 在提高判别模型通用性的问题上，使用 L2 正则也可以起到效果；同时，较高的 L2 正则还可以降低判别模型的能力，让生成模型变得更加容易学习。

在另外一篇文章 *Improved Techniques for Training GANs*^[3] 中，作者提到了提高模型能力的其他方法。前面提到了生成模型“不思进取”的现象，当生成模型找到可以混淆判别模型的输出图像后，为了保证目标函数足够小，它有可能迅速“崩塌”，将所有的输入都映射到这些图像上。这样虽然最终的目标函数得到了满足，但是这个生成模型的实际能力并不强大，也不是我们想要的效果。这说明在训练过程中采用之前的损失函数是不够的，还需要为模型增加更多的约束。基于这样的思想，文章的作者又添加了新的约束。

在新的约束中，作者希望真实数据的中间特征和生成数据的中间特征能够足够相近。这样图像的生成难度就增大了很多，除要求图像最终被判别模型认可外，还需要图像的中间表达和这一批次训练的数据足够接近。这样生成模型就很难将所有输入映射到同一个输出上了，同时，多了一个约束也让优化目标变得更加清晰，优化的过程也更加稳定。这个优化目标如下：

$$\|E_{x \sim p_{\text{data}}} f(x) - E_{z \sim p_z(z)} f(G(z))\|_2^2$$

除了这个约束，文章中还介绍了很多优化的细节方法，这些方法和优化过程中常用到的一些方法比较相近，这里不再赘述。

28.4 InfoGAN——探寻隐变量的内涵

上面介绍了 GAN 的一些基本原理和优化细节，接下来将介绍更多生成模型分布的细节。采用前面介绍的方法，生成接近于真实的图像已经成为可能，但是这似乎还不够。现在构建生成模型的方法和传统方法不同，在传统方法中，每一个隐变量都会有一些特定的含义，或者能够被发现一些直观的与生成结果的关系，而现在采用高度复杂的非线性函数做拟合，原始隐变量变得十分简单，简单到似乎不太容易找到它与生成数据的关系。比如想要生成一张人脸，我们无法很好地控制生成人脸的细节：脸型、眼睛、鼻子、嘴等，GAN 只能帮助生成一样接近于真实的脸，却不能帮助做更多定制化的工作。

基于这样的问题, 论文 *InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets*^[4] 中提出了一种全新的目标函数, 这种目标函数可以很好地兼顾原来的优化目标, 同时解决定制化生成的问题。

InfoGAN 方法的第一步是重新审视隐变量。之前的隐变量主要产生随机噪声的作用, 确保隐变量分布可以扩展成真实数据的分布, 这里面对隐变量的职责并没有过多的约束; 现在隐变量除拥有随机噪声的作用之外, 还拥有控制生成结果的作用, 为此隐变量将被分离成两个部分。

- z : 继续充当随机噪声的作用。
- c : 充当隐式变量的作用, 影响最终生成的数据。

这样曾经的生成模型 $G(z)$ 就变成了 $G(z, c)$ 。既然知道了 c 和 G 的关系, 那么可以想象 c 和 G 的信息是非常相关的, 于是它们之间的关系就可以用互信息 $I(c; G(z, c))$ 这个指标表示出来, 而且它们的互信息值会很高。

互信息 (Mutual Information) 是信息论中衡量随机变量关系的一种指标, 如果有两个随机变量 X 和 Y , 那么它们的互信息表示两个随机变量相互关联的信息量。大家都知道, 在信息论中熵是衡量随机变量信息量的指标, 互信息的量就是一个随机变量 X (或 Y) 的熵减去已知另一个随机变量 Y (或 X) 后它的熵:

$$I(X; Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

从公式中可以看出, 如果知道随机变量 X 后, Y 的信息量为 0, 那么两者的互信息值将会非常高。从这个概念上看, c 和 G 的互信息值应该非常高, 所以要想让 c 起到控制生成图像的效果作用, 这部分就可以加入到目标函数中, 由此目标函数就变成了:

$$\min_G \max_D V_I(D, G) = V(D, G) - \lambda I(c; G(z, c))$$

虽然加入互信息的目标函数从原理上讲可以起到相应的作用, 但是这个指标并不是一个容易优化的函数, 为此需要对公式做一定的变换, 论文中采用了 Variational Inference 的方法, 最终的目标函数中的互信息部分变为:

$$E_{c \sim P(c), x \sim G(z, c)} [\log Q(c|x)] + H(c)$$

这个公式可以为判别模型和生成模型提供梯度, 只要对公式做相应的变换即可。

在实验过程中, 模型 Q 和 D 的大部分参数是重合的, InfoGAN 模型架构如图 28-5 所示。

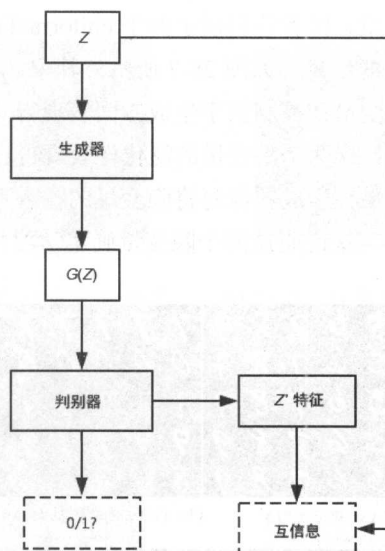
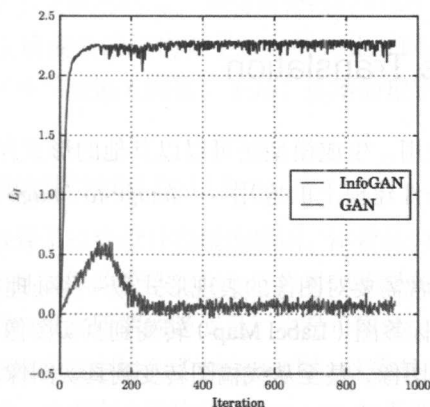


图 28-5 InfoGAN 模型架构图

为了展示模型学到的效果，进行了一系列实验。首先是 MNIST 数据集，作者在隐变量集合中加入了一个均匀分类分布（Uniform Categorical Distribution），这个分布的维度为 10，每一维出现的概率为 0.1，我们希望通过学习它能够学习到数据表示的数字信息。从图 28-6 所示的实验结果可以看出，模型的互信息值非常高，模型确实学习到了隐变量和图像之间的关联关系；相比之下，没有互信息目标的 GAN 模型则没有达到这样的效果。通过这个实验可以说明，互信息目标确实帮助传统的 GAN 模型拥有了可表达的隐变量，生成模型可以利用隐变量完成更多的工作。

图 28-6 InfoGAN 和 GAN 在互信息上的对比^[4]

除了表示数字内容的隐变量,作者还创建了两个 uniform 的连续隐变量 c_2, c_3 , 希望这两个隐变量能够挖掘出更多隐藏的信息。如图 28-7 所示, 其中 c_2 表示数字旋转的角度, c_3 表示数字笔画的粗细。这两个隐变量在控制数字生成方面表现得非常自然, 让人不太容易看出其中不自然的地方。与此同时, 这两个隐变量的泛化性表现得也非常好, 虽然在模型中限定变量的范围为 $(-1, 1)$, 但是如果在生成图像时将隐变量的输入范围扩展到 $(-2, 2)$, 那么生成的结果同样合理且符合预期——这说明这两个隐变量确实学习到了高质量的内部信息。

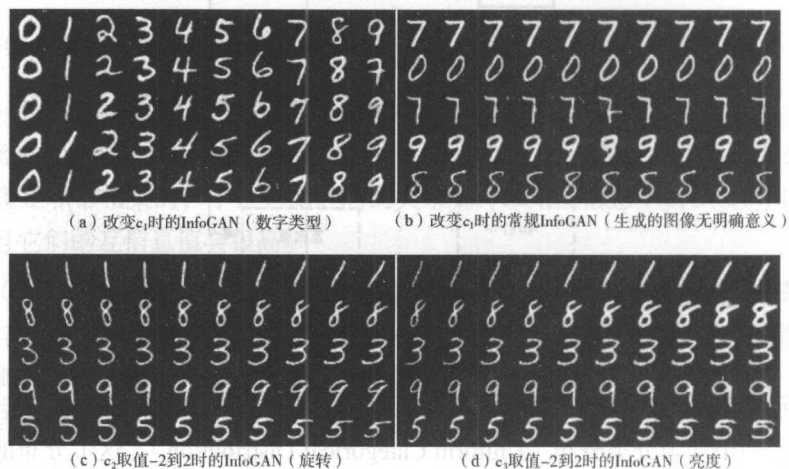


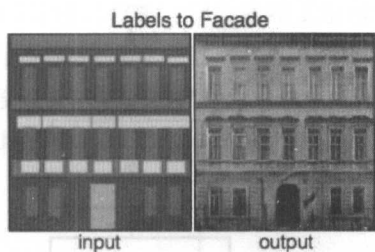
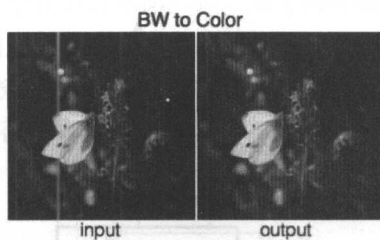
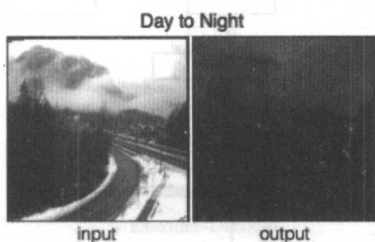
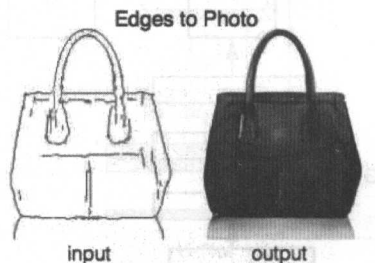
图 28-7 模型学习到的隐含变量效果^[4]

除此之外, 作者还将这个实验应用在其他的数据集上, 模型在这些数据集上的表现都非常好, 这说明了方法的有效性。实际上利用学习到的隐变量信息, 我们可以对生成数据做更多的分析, 未来针对数据的应用也有更多的可能。

28.5 Image-Image Translation

除了前面几节介绍的应用, 生成模型还可以以其他的形式存在, 比如对条件分布建模。本节将介绍 GAN 模型在条件建模上的应用——*Image-to-Image Translation with Conditional Adversarial Networks*^[5]。

在实际应用中, 我们经常需要对图像的表现形式做一些处理, 同时还要确保图像在语义上保持不变。比如从物体的标签图 (Label Map) 转变到真实图像、从黑白图像转变到彩色图像、从白天图像转变到夜晚图像, 甚至从线稿图转变到真实图像, 分别如图 28-8 至图 28-11 所示。

图 28-8 从标签图转变到真实图像^[5]图 28-9 从黑白图像转变到彩色图像^[5]图 28-10 从白天图像转变到夜晚图像^[5]图 28-11 从线稿图转变到真实图像^[5]

从这些图像中可以看出，图像的变换有一定的特点。首先，图像在变换时保持了语义的一致性，原图描述的物体和转变后的图像描述的物体是一致的；其次，图像的变换往往不是唯一解，实际上变换图像可以有很多选择，这些不同的选择可以用一个概率分布去描述。

基于上面的分析，问题可以表述成寻找一个映射，当给定原始图像 X 、隐变量 Z 时，求出转变图像 Y ：

$$X, Z \rightarrow Y$$

这个映射就是生成模型要完成的内容。和前面介绍的 GAN 模型不同，这一次的模型包含了一张输入图像，因此生成模型的结构有了很大的变化。同时，为了方便判别模型进行判断，判别模型需要加入另外一张输入图像。于是，模型的理想结构变成了图 28-12 所示的样子。

但是实际上，本文采用的模型如图 28-13 所示。

作者在论文中也简单解释了这样设计模型的原因。作者认为学习加入隐变量的模型并不一定会带来更好的效果，而且这样的模型实际上已经可以满足需求：从原始图像到目标图像的生成。即使只是一个映射，没有所谓的隐变量，也是可以的。当然，为了让模型拥有一些随机性，作者在模型的部分网络层中加入了 Dropout 层，这样具备一定 Ensemble 能力的模型会增加一定的变化性（不过，作者在文章中表示 Dropout 对表现力增加的贡献有限）。

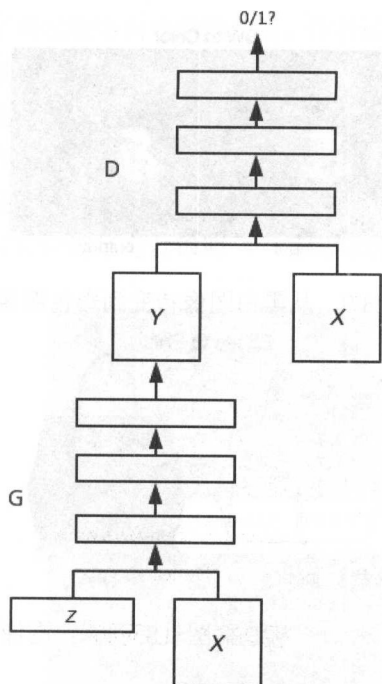


图 28-12 理想的模型结构

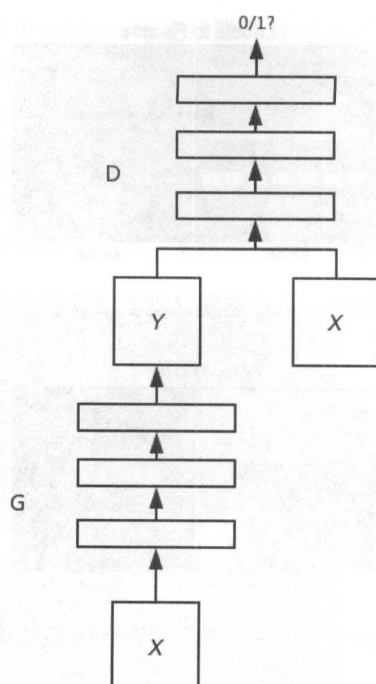


图 28-13 pix2pix 模型结构

这个模型结构使得 Generator 部分的结构也变得相对简单一些，由于输入和输出的维度一致，因此模型可以用全卷积网络来表示。按照全卷积网络的形式，随着模型的计算，一开始图像特征的长度和宽度将变小，同时通道的数量会增多；当网络达到一定程度时，特征的长度和宽度将变大，同时通道的数量减少，最终到达输出层，结果的维度和输入的维度保持一致。根据图像特征在网络结构中的变化情况，这种网络也被称为“Encoder-Decoder”网络，如图 28-14 所示。

虽然这样的网络可以满足要求，但是网络对于变换过程缺少了一些约束。实际上在图像变换的过程中，我们需要图像的整体结构保持不变，物体的轮廓信息保持不变。因此，如果对模型增加一些约束，它的表现会更好一些，模型收敛也会更容易一些。于是，作者在“Encoder-Decoder”网络的基础上增加了部分结构，使得图像长、宽变小的参数被再次利用到模型上，此时的模型结构被称为 U-网络，如图 28-15 所示。

这时模型就可以更好地完成物体表面变换和物体结构保持这两件事情了。

同样地，对于判别模型，它的目标是判定生成模型转换的效果，这个转换效果相对而言更偏向于细节部分，因此判别模型的评估也需要做出一定的改变，不能像之前的模型那样，

对整个图片做出评价。作者采用了给每个局部一个评价概率的形式，对于两张输入的图像，首先将它们按通道维度拼接起来，然后再对它们进行卷积处理，使图像的长、宽维度变小，最终达到 $1 \times N \times N$ 的维度。此时的 N 代表了长度和宽度，也就是说，整个图像被分成了许多小块，判别模型将评价每个小块内的图像是否一致。

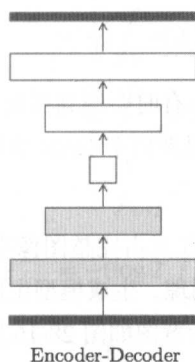


图 28-14 “Encoder-Decoder” 网络^[5]

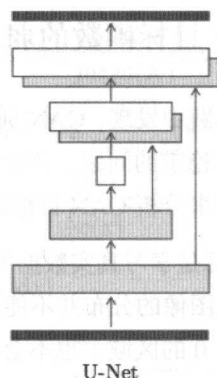


图 28-15 U-网络^[5]

这样带来了一定的好处，使判别模型更关注局部，同时也使得生成模型的变换效果的量化更为细致。如果观察整张图片的变换效果，由于生成模型未达到最优，效果可能不会很好，那么判别模型可以比较容易地将生成图像和真实图像区别开；而使用了局部匹配的方式，模型在局部优异的生成效果便会得到肯定，这样会得到好的效果。这个网络也被作者称为“PatchNet”，因为局部匹配的目标是图像中的一个一个小部分。

这两项改进是否会增强模型的效果呢？为了回答这个问题，作者进行了一些实验尝试。对于 U-网络的架构形式来说，生成模型可以产生更好的图像，同时保证局部细节的锐利；而普通的“Encoder-Decoder”网络在细节上稍显逊色。另外，利用 Patch 进行比较既可以保证图像的细节效果，同时又可以增加生成模型收敛的速度。关于实验部分的内容，请有余力的读者自行阅读论文去体会改进的效果。

28.6 WGAN (Wasserstein GAN)

从 GAN 诞生之际开始，很多科研人员就在努力试图增强它的能力。这里面有两个原因，首先，GAN 确实达到了非常好的效果，让人们找到了构建复杂生成模型的新希望；其次，GAN 确实存在一些问题，有时生成模型无法收敛，有时生成模型的表现不稳定，这些都对 GAN 的使用提出了挑战。针对这些挑战，科研人员对模型做了很多修改，有的修改

了网络的结构，有的修改了网络的目标函数，还有的对两者都做了修改。接下来的这篇文章——*WassersteinGAN* 也是其中之一，不同的是它对 GAN 的目标函数做了深入分析，提出了结构相近但效果提升明显的模型。

28.6.1 GAN 目标函数的弱点

很多人在实践中发现，GAN 训练过程并不总是很顺利，有时会遇到梯度消失的问题，有时会遇到优化不稳定的问题。产生这些问题的一个重要原因来自于 GAN 的目标函数，下面就从理论的角度来分析 GAN 目标函数存在的一些问题。

GAN 的目标是学习真实数据的分布，然而真实数据往往只占完整图像空间的一小部分，也就是说，真实图像的分布并不能占满完整的图像空间。同理，生成模型生成的分布的支撑面（概率值不为 0 的区域）也不会占满整个图像空间。其示意图如图 28-16 所示。

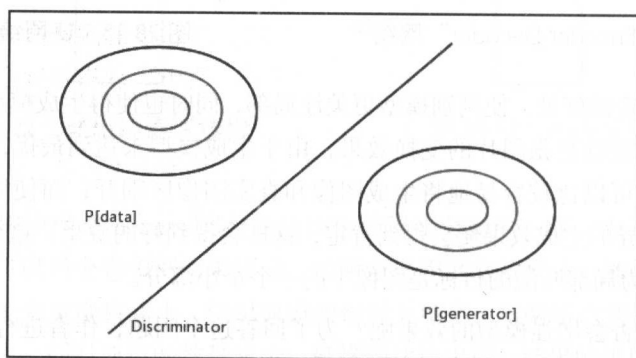


图 28-16 真实图像分布和生成图像分布在完整图像空间中的示意图

这样的分布形式会存在什么样的问题呢？那就是真实数据分布和生成数据分布的支撑面没有交集或者交集很小。读者可以回顾曾经接触过的 KL 散度定义：

$$KL(p||q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

实际上，KL 散度的计算只在两个分布支撑面重合的区域有意义，在其他区域得到的结果并没有意义。对于上面的公式，如果存在某个 x ，使得 $p(x) \neq 0, q(x) = 0$ ，那么 KL 值将变为无穷大；如果反过来， $p(x) = 0, q(x) \neq 0$ ，那么 KL 值将等于 0。显然这两个值都不是我们希望得到的数值。所以在两个分布不重叠时，求解 KL 散度实际上是没有意义的。

当然，前面的章节曾提到，在 GAN 中生成模型的优化目标实际上是 JS 散度，这种对称的散度比 KL 散度效果更好。那么它在这种不重叠的分布上表现效果如何？假设有两个概率分布 $p(x)$ 和 $q(x)$ ，这两个分布的支撑面不相交，那么有：

$$\begin{aligned}
 JS(p||q) &= \frac{1}{2}(KL(p||\text{mean}) + KL(q||\text{mean})) \\
 &= \frac{1}{2}(\int_X p(X) \log \frac{p(X)}{\text{mean}(X)} dX + \int_X q(X) \log \frac{q(X)}{\text{mean}(X)} dX) \\
 &= \frac{1}{2}(\int_{x \sim P} p(x) \log \frac{p(x)}{\text{mean}(x)} dx + \int_{x \sim Q} q(x) \log \frac{q(x)}{\text{mean}(x)} dx) \\
 &= \frac{1}{2}(\int_{x \sim P} p(x) \log \frac{p(x)}{\frac{1}{2}p(x)} dx + \int_{x \sim Q} q(x) \log \frac{q(x)}{\frac{1}{2}q(x)} dx) \\
 &= \frac{1}{2}(\log 2 \int_{x \sim P} p(x) dx + \log 2 \int_{x \sim Q} q(x) dx) \\
 &= \log 2
 \end{aligned}$$

通过计算可以看出，只要两个分布的支撑面不相交，JS 散度的测量值就为常值，这个数值同样不令人满意，这很难衡量出两个分布之间的距离。因此，获得梯度也变得十分困难。

从目标函数的计算来看，当两个分布的支撑面没有重叠时，GAN 的优化可能存在问题；从对抗的角度来看，没有重叠同样会出现问题。一旦两个分布没有重叠，对于一般使用的完备正规空间来说，根据乌雷松引理，找到一个连续函数将两个支撑面的映射值完美分开是可行的。而一旦判别模型可以完美分开生成数据分布和真实数据分布，那么生成模型的梯度将消失，这样模型的训练将无法进行。

正是因为这个原因，我们在训练 GAN 模型时需要对判别模型的训练做一定的权衡——既不能让判别模型太强，这样可能会使生成模型无法训练；同时也不能让判别模型太弱，否则生成模型将无法学到和真实数据近似的分布。

既然 GAN 的目标函数存在一定的劣势，那么我们需要将其更换成什么函数呢？

28.6.2 Wasserstein 度量的优势

Wasserstein 度量 (Metric)，作为最优运输理论中的一个经典度量，它表示在一个度量空间下两个概率分布之间的距离。它和 KL 散度、JS 散度一样，都用来描述分布间的差异，但是它们的表述形式是不同的。Wasserstein 距离的公式如下：

$$W(\mu, \nu) = \inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} d(x, y) d\gamma(x, y)$$

这个公式表示的是 1 阶 Wasserstein 距离。其中 μ 和 ν 各自表示完备正规空间 X 、 Y 下的概率测度函数。我们定义一个拓扑积空间 $Z: X \times Y$ ，这个是最终要考虑的空间。另外定义两个投影操作 pr_X 和 pr_Y ，它们的作用是将 Z 空间分别投影到 X 、 Y 空间。接下来就要定义 Z 空间下的概率测度函数 π ，这个函数除了需要满足概率测度函数应该满足的性质，还需要满足下面两个公式：

$$pr_X(\pi) = \mu$$

$$pr_Y(\pi) = \nu$$

所有满足这些条件的概率测度函数组成了一个集合 $\Gamma(\mu, \nu)$ ，我们的目标是找出其中的一个概率测度函数，使得等式右边的积分最小。

那么积分中的内容是什么含义呢？对于 Z 空间中的每一个点，我们都要计算这个点的两个投影下的点的距离 $d(x, y)$ 。这个距离可以想象成将 x 点移动到 y 点的距离。考虑到整个乘积拓扑空间，就是找到一种点与点之间的匹配方法，使得将一个支撑面移动到另一个支撑面的代价最小。由于支撑面上每一个点的概率测度不同，它们移动的代价也不同，因此移动时需要考虑乘积拓扑下的概率测度。该描述实际上依然十分晦涩，下面我们将使用论文中提到的一个例子来分析 Wasserstein 距离的具体计算方法。

论文中的例子希望计算二维欧式空间下两个子空间的概率测度函数之间的距离。其中有一个随机变量： z ，服从 $(0, 1)$ 的均匀分布。

需要计算的概率测度函数的支撑面分别是：

$$X: \{x|x = (0, z)\}$$

$$Y: \{x|x = (\theta, z)\}, \forall \theta \in R$$

两个函数所表示的分布均为均匀分布。

可以看出，当 $\theta = 0$ 时，两个分布的支撑面完全重合，在其他情况下两者完全不重合。根据上一节的分析，当两个分布的支撑面完全不重合时，KL 散度的计算结果为正无穷大，而 JS 散度的距离为 $\log 2$ ，这两种度量显然不能准确描述两个函数之间的距离，由这两种度量诱导出的拓扑空间在性质上也会差些。那么 Wasserstein 度量呢？

我们首先将两个概率测度函数进一步形式化，它们可以写作：

$$X : p(x) = 1$$

$$Y : p(y) = 1$$

由于要确定待移动的点信息，所以将所有明确的移动点对定义为 $\text{pair}(x, y)$ ，这样积空间的测度函数就可以定义为：

$$Z(X \times Y) : p(x, y) = 1(x, y) \in \text{pair}(x, y)$$

那么这个概率测度函数是否属于 $\Gamma(\mu, \nu)$ 呢？读者可以自行验证上面提出的那些条件，看看这个测度函数是否满足。

下面就要求解 Wasserstein 距离了。当然，由于问题比较简单，我们可以通过观察图像发现，如果把支撑面 Y 想象成一条线段，那么 Wasserstein 距离就是求把 Y 平移到 X 所经过的面积，如图 28-17 所示。

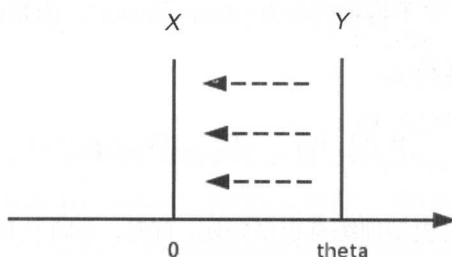


图 28-17 Wasserstein 距离示意图

根据上面的分析，复杂的 Wasserstein 距离可以化简成下面的公式：

$$W(\mu, \nu) = |\theta|$$

这个结果和 KL、JS 有很大的不同，它不是阶梯状的函数，同时它是一个可导的函数，这个距离变得更利于计算了。这只是它带来的一个表面的好处，实际上它能够保证模型优化更加稳定，收敛速度也更有保证。

28.6.3 WGAN 的目标函数

实际上,上面的例子比较简单、直观,我们采用直接观察图像的方式求解出最终的结果,但是在 GAN 中,两个生成模型非常复杂,采用上面的形式进行计算会非常复杂且不可导,因此在正式计算时需要对目标函数进行一定的转化。转化的过程涉及 Kantorovich-Rubinstein 对偶原理和 Monge-Kantorovich 对偶原理等知识,证明过程相对复杂,这里不再赘述。最终目标函数化简为:

$$W(\mu, \nu) = \sup_{\|f\|_L \leq 1} E_{x \sim \mu}[f(x)] - E_{x \sim \nu}[f(x)]$$

公式中的 f 有一个附加限制条件,那就是必须要满足 1-Lipchitz 条件。也就是说,函数的这个条件十分重要,只有满足这个条件,目标函数的化简才能达成。但是它也给优化带来一点麻烦,为了尽量不限制优化, f 函数被适当放松,只要满足 K -Lipchitz 即可,这相当于为目标函数乘以一个系数,实际上并不会改变最优解出现的位置。而且在优化时函数不做任何限制,每次优化完成后,模型参数都会被截断到一定的范围内,以确保满足限定条件。由于判别函数变成了上面的形式,它已经不再用于判别图像是否属于真实数据了,而变成了一个度量的形式,因此在论文中不再将其称为“discriminator”,而是改名为“critic”。

最终生成模型的梯度变为:

$$\nabla_{\theta} W(P_r, P_{\theta}) = -E_{z \sim p(z)}[\nabla_{\theta} f(g_{\theta}(z))]$$

使用 Wasserstein 距离对模型训练有很多好处。首先,不同于 JS 散度, Wasserstein loss 是连续可导的,这样即使真实数据和生成数据的分布不重合,判别模型将二者完美分开,生成模型也依然拥有梯度,并不会遇到梯度饱和的问题;其次, Wasserstein 距离直接衡量了两个分布的移动距离,从数值上看更容易理解。这两点就是 WGAN 与之前模型相比的优势,论文中还给出了几个例子,这里不再赘述,欢迎感兴趣的读者自行阅读。WGAN 的算法如下:

算法 28-2 WGAN 算法

算法用到的变量

α : 学习率

c : 梯度裁剪值

m : 一批训练数据的数量

n_{critic} : 每轮整体迭代中 critic 独自的迭代轮数

w_0 : critic 的初始参数值

θ_0 : 生成模型的初始参数值

while θ 没有完成收敛 **do**

for $t = 0, \dots, n_{\text{critic}}$ **do**

从真实数据中根据概率分布 P_r 批量采样数据 $\{x^{(i)}\}_{i=1}^m \sim P_r$

根据先验概率分布 $p(z)$ 采样数据 $z^{(i)}_{i=1}^m$

$$g_w \leftarrow \nabla_w \left[\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)})) \right] w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w) w \leftarrow \text{clip}(w, -c, c)$$

end for

根据先验概率分布 $p(z)$ 采样数据 $\{z^{(i)}\}_{i=1}^m$

$$g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$$

$$\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$$

end while

参考文献

- [1] Goodfellow I J, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets. International Conference on Neural Information Processing Systems. MIT Press, 2014:2672-2680.
- [2] Radford A, Metz L, Chintala S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Computer Science, 2015.
- [3] Salimans T, Goodfellow I, Zaremba W, et al. Improved Techniques for Training GANs. 2016.
- [4] Chen X, Duan Y, Houthoofd R, et al. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. 2016.
- [5] Isola P, Zhu J Y, Zhou T, et al. Image-to-Image Translation with Conditional Adversarial Networks. 2016.
- [6] Arjovsky M, Chintala S, Bottou L. Wasserstein GAN. 2017.
- [7] Arjovsky M, Bottou L. Towards Principled Methods for Training Generative Adversarial Networks. 2017.
- [8] Villani C. Optimal transport: old and new. 2008, 338.

A

本书涉及的开源资源列表

章节	资源名称	资源位置
8.1	Theano	http://deeplearning.net/software/theano/
8.2	Torch	http://torch.ch/
8.3	PyTorch	http://pytorch.org/
8.4	Caffe	http://caffe.berkeleyvision.org/
8.5	TensorFlow	https://www.tensorflow.org/
8.6	MXNet	http://mxnet.io/
8.7	Keras	https://keras.io/
10.1	LeNet-5	http://yann.lecun.com/exdb/lenet/
10.2	AlexNet	https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet
10.3	VGGNet	http://www.robots.ox.ac.uk/~vgg/research/very_deep/
10.4	GoogLeNet	https://github.com/tensorflow/models/tree/master/inception
10.5	ResNet	https://github.com/KaimingHe/deep-residual-networks
10.6	DenseNet	https://github.com/liuzhuang13/DenseNet
10.7	DPN	https://github.com/cypw/DPNs
11.1.2	OverFeat	https://github.com/sermanet/OverFeat
11.2.1	R-CNN	https://github.com/rbgirshick/rcnn
11.2.2	SPP-net	https://github.com/ShaoqingRen/SPP_net

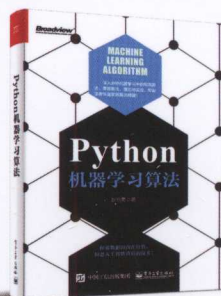
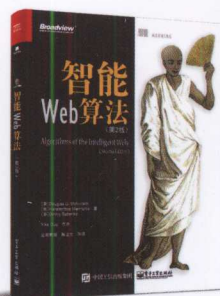
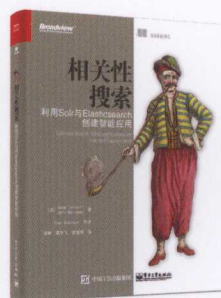
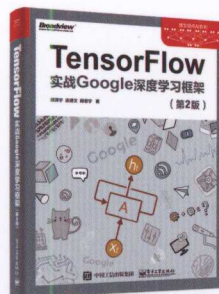
续表

章节	资源名称	资源位置
11.2.3	Fast R-CNN	https://github.com/rbgirshick/fast-rcnn
11.2.4	Faster R-CNN	https://github.com/ShaoqingRen/faster_rcnn
11.2.5	R-FCN	https://github.com/daijifeng001/R-FCN
11.3.1	YOLO	https://pjreddie.com/darknet/yolo/
11.3.2	SSD	https://github.com/weiliu89/caffe/tree/ssd
12.1.1	FCN	https://github.com/shelhamer/fcn.berkeleyvision.org
12.1.2	DeconvNet	https://github.com/HyeonwooNoh/DeconvNet
12.1.3	SegNet	http://mi.eng.cam.ac.uk/projects/segnet/
12.1.4	DilatedConvNet	https://github.com/fyu/dilation
12.2.1	DeepLab	http://liangchiehchen.com/projects/DeepLab.html
12.2.2	CRFasRNN	https://github.com/torrvision/crfasrnn
12.2.3	Deep Parsing Network	http://personal.ie.cuhk.edu.hk/~lz013/projects/DPN.html
12.3.1	Mask R-CNN	https://github.com/CharlesShang/FastMaskRCNN (非原作)
13.3	DSH	https://github.com/lhmRyan/deep-supervised-hashing-DSH (非原作)
15	OpenFst	http://www.openfst.org/
16.2	Kaldi	http://kaldi-asr.org/
16.4	EESEN	https://github.com/yajiemiao/eesen
19.2	Stanford CoreNLP	https://stanfordnlp.github.io/CoreNLP/
19.3	JNN	https://github.com/wlin12/JNN
20.2	SyntaxNet	https://github.com/tensorflow/models/tree/master/syntaxnet
21	word2vec	https://code.google.com/archive/p/word2vec/
21.6	fastText	https://github.com/facebookresearch/fastText
21.7	GloVe	https://nlp.stanford.edu/projects/glove/
22.2	GroundHog	https://github.com/pascanur/GroundHog
22.4	GNMT	https://github.com/tensorflow/nmt
22.5	FAIRSeq	https://github.com/facebookresearch/fairseq
25.6.1	TCDCN	http://mmlab.ie.cuhk.edu.hk/projects/TCDCN.html
25.6.2	DeepID2	https://github.com/happynear/FaceVerification (非原作)

续表

章节	资源名称	资源位置
25.6.5	MNC	https://github.com/daijifeng001/MNC
26.4	HashedNets	http://www.cse.wustl.edu/~wenlinchen/project/HashedNets/index.html
26.5	Squeeze-Net	https://github.com/DeepScale/SqueezeNet
26.6	BinaryConnect	https://github.com/MatthieuCourbariaux/BinaryConnect
26.6	BinaryNet	https://github.com/MatthieuCourbariaux/BinaryNet
26.7	MobileNet	https://github.com/tensorflow/models/blob/master/slim/nets/mobilenet_v1.md
27.5	DQN	https://deeppmind.com/research/dqn
28.3	DCGAN	https://github.com/carpedm20/DCGAN-tensorflow
28.4	InfoGAN	https://github.com/openai/InfoGAN
28.5	Pix2Pix	https://github.com/phillipi/pix2pix
28.6	WGAN	https://github.com/martinarjovsky/WassersteinGAN

好书分享



深度学习

核心技术与实践

拍照搜题APP“小猿搜题”，以及猿辅导公司一系列被称为“小猿黑科技”的产品——英语作文自动批改、英语口语自动打分纠错、速算应用中的在线手写识别等的核心部分，都是我们的应用研究团队，也就是本书的作者们实现的。在几乎全经济部门言必称人工智能、深度学习之时，出版这样一线业者的著作，是真正有益的工作。一个公司所做的，不仅有益于用户，也能有益于行业，本书的出版也是我司的骄傲时刻。感谢应用研究团队。

猿辅导公司CEO 李勇

本书的作者之一邓澍军博士和夏龙是我的老同事，几年前我们在网易有道共事时，他俩就开始了孜孜不倦的机器学习“修炼”之旅，读经典专著和论文，研读代码，推动机器学习技术和公司业务结合，这股劲头一直延续到他们加入猿辅导创业。今天，他们把自己对深度学习方法的心得体会、落地的第一手经验凝集在《深度学习核心技术与实践》这本书里，即使是我这种自认为经验很丰富的人，也从这本书中学到了很多不曾了解的知识。

北京一流科技有限公司创始人 袁进辉（老师木）

这本书的不少作者都是我的前同事。从书中我看到了熟悉的务实、钻研、追求实际效果的风格。在深度学习被称为“炼金术”的当前，本书通过第一线的视角，既包含工程实践所需的关键概念、模型和算法原理，也有多年实践经验的总结。本书内容深入浅出，干货满满，是一本不可多得的入门和实践参考书。

网易有道首席科学家 段亦涛



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



策划编辑：张春雨
责任编辑：葛娜
封面设计：李玲

上架建议：人工智能>深度学习

ISBN 978-7-121-32905-0



9 787121 329050 >

定价：119.00元